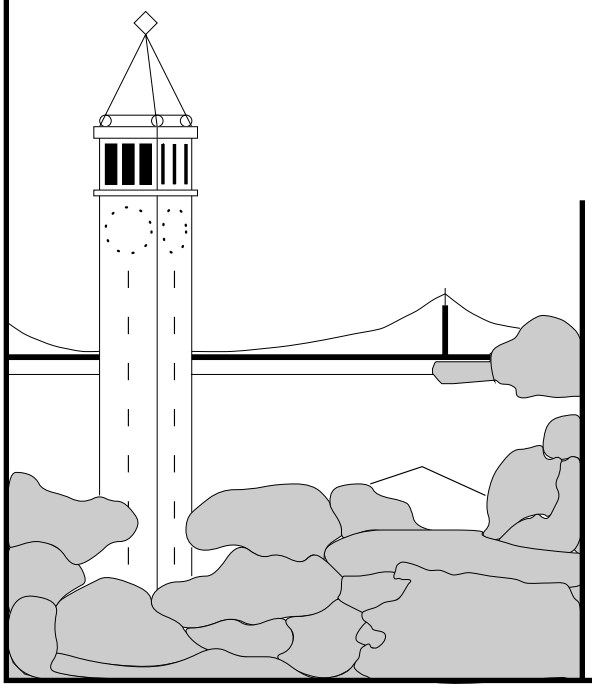
Multivalent Documents: Anytime, Anywhere, Any Type, Every Way User-Improvable Digital Documents and Systems

Thomas A. Phelps



Report No. UCB/CSD-98-1026

December 1998

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Multivalent Documents:
Anytime, Anywhere, Any Type, Every Way
User-Improvable
Digital Documents and Systems

by

Thomas Arthur Phelps

B.S. (University of Illinois at Urbana-Champaign) 1990
A.B. (University of Illinois at Urbana-Champaign) 1990
M.S. (University of California, Berkeley) 1993

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Robert Wilensky, Chair
Professor Lawrence A. Rowe
Professor Mary Kay Duggan
Dr. Gary E. Kopec

1998

This research was funded as part of the NSF/NASA/DARPA Digital Library Initiative, under National Science grant number IRI-9411334.

Multivalent Documents:
Anytime, Anywhere, Any Type, Every Way
User-Improvable
Digital Documents and Systems

Copyright © 1998 by Thomas Arthur Phelps
All rights reserved.

An online copy of the document can be found via
http://www.cs.berkeley.edu/

Abstract

Multivalent Documents:
Anytime, Anywhere, Any Type, Every Way
User-Improvable
Digital Documents and Systems

by Thomas Arthur Phelps

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Robert Wilensky, Chair

Digital documents are important. Whatever else computer workers do, they expend a considerable time working with digital documents, whether as e-mail, word processing files, presentation slides, web pages, discussion groups, or help systems, among many other ways. This dissertation shows how to improve the online manipulation capabilities of potentially all formats, media types, and genres of existing and future digital documents.

The *Multivalent Document Model* extensively opens to enhancement all aspects of a digital document system. Document content is constructed from *layers* of often heterogeneous type, each with specialized purpose, all semantically aligned. All user-visible document functionality is constructed from stylized program components called *behaviors*. Document system operations, such as drawing a representation of the document on the screen and saving an edited version, derive from the fundamental operation found to some degree in every digital document system, newly codified as extensible programmatic protocols. This diverse open content, open functionality, and open operation are woven together by numerous mechanisms to produce a final composition that appears built from the ground up as a unified whole.

A prototype of the Model, called the *Multivalent Document System*, has been realized in Java, deployed, and built upon by several third party developers. The System has been tested by and has significantly contributed to the development of three sample applications. The first application allows paper scanned into a computer as images to be manipulated as a live, semantic object, with text copy and paste, text search, and a "lens" operation that displays the corresponding ASCII translation of the region. In the second

application, HTML, the lingua franca of the web and a very different document type than scanned page images, has been extended with new functionality including outline displays, a speed reading window, and tables sorted on demand. As a third application, both of the above document types can be annotated in situ with hyperlinks, highlights, floating note windows, a new display mode called *Notemarks*, and executable copy editor markup. Annotations and behaviors in general can be distributed across the network, augment documents on read-only media, and operate on potentially any document format with a single, format-neutral implementation.

This dissertation describes the design of the Multivalent Document Model, its implementation as the Multivalent Document System, and the specialization of the Model in each of the three example applications.

To my parents who begat me.

Anything you do,
Let it come from you.
Then it will be new.

Give us more to see…

— Stephen Sondheim
Sunday in the Park with George

Contents

1 Introduction **1**

1.1 Problems: Documents Are Diverse and Not Adaptable or Progressive *1*
1.1.1 Mediocrity through Diversity 1
1.1.2 Documents and Systems Not Adaptable or Progressive toward Future 3

1.2 Goal: Extremely Improvable Digital Documents and Systems *5*

1.3 A Solution: Pervasively Open, Highly Factored Documents and Systems *7*
1.3.1 Demonstration 8
1.3.2 Architecture 15

1.4 Contributions *21*

1.5 Overview of Dissertation *22*

2 Opening the Fundamental Document Lifecycle as Protocols **24**

2.1 Fundamental Document Lifecycle *24*

2.2 Extensibility and Composition through Open Protocols *25*
2.2.1 General Schematic 26
2.2.2 General Schematic of Tree Protocols 27
2.2.3 Summary 29

2.3 Multivalent Protocols in Detail *30*
2.3.1 Restore Protocol 31
2.3.2 Build Protocol 34
2.3.3 Format Protocol 41
2.3.4 Paint/Print Protocol 44
2.3.5 Events Protocol 47
2.3.6 Clipboard Protocol 53
2.3.7 Undo/Redo Protocol 57
2.3.8 Save Persistent State Protocol 58
2.3.9 Protocol Summary 59

3 Highly Factoring Document Functionality and Content as Behaviors and Layers 61

3.1 Behaviors *61*
3.1.1 Media Adaptors 62
3.1.2 Structural 64
3.1.3 Spans 65
3.1.4 Geometric/Lens 67
3.1.5 Managers 69

3.2 Layers of Information *69*
3.2.1 Content Layer: Unit of Document Construction 70
3.2.2 Hierarchical Collection 72

4 Supporting Architecture 73

4.1 Document Kernel *73*

4.2 The Document Tree *75*
4.2.1 General Tree Node Properties 75
4.2.2 Abstract Structural Tree Embodied in Internal Nodes 76
4.2.3 Special Cases Managed by Root Node 78
4.2.4 Medium-dependent Leaves Enable Medium-independent Behaviors 79

4.3 The Graphics Context *79*
4.3.1 Standard Properties 80
4.3.2 Setting Property Values 81

4.4 Persistent Manifestation: Hub Document *82*
4.4.1 Cascading and Spontaneous Hubs 82
4.4.2 Sample Hub Document 84

4.5 General System Support *87*
4.5.1 Picking 87
4.5.2 Grabs 87
4.5.3 Style Sheet 87
4.5.4 Global State 88
4.5.5 Editing 90

5 Implementation: Performance, Scalability, Robustness 91

5.1 Performance and Scalability through Pervasive Incrementality *92*
5.1.1 Efficient Computation of Active Behaviors at Given Point in Tree 92
5.1.2 Incremental Formatting for Editing and Fast Startup 96
5.1.3 Incremental Painting for Scrolling, the Selection, and Lenses 98
5.1.4 Performance Measurements 99
5.1.5 Remaining Performance Issues 103

5.2 Robust Locations *104*
5.2.1 Location Types 105
5.2.2 Comparison 109

6 Applications 110

6.1 Enlivening Scanned Page Images *111*
6.1.1 Problem 111
6.1.2 Solution 112
6.1.3 Implementation: Specializing the Multivalent Protocols 113
6.1.4 Related Work 118

6.2 Extensible HTML *120*
6.2.1 Problem 120
6.2.2 Solution 121
6.2.3 Implementation: Specializing the Multivalent Protocols 123
6.2.4 Related Work 125

6.3 Distributed Annotations In Situ *126*
6.3.1 Problem 126
6.3.2 Solution 127
6.3.3 Novel Annotation Type: Notemarks 131
6.3.4 Implementation: Specializing the Multivalent Protocols 132
6.3.5 Related Work 134

7 Related Work **136**

7.1 Integration vs. Juxtaposition: OpenDoc, OLE, Quill, HTML with Java Applets and Plug-ins *136*
7.1.1 OpenDoc 138
7.1.2 OpenDoc Protocols 139
7.1.3 Quill Protocols 140

7.2 Composition vs. Shared Library: GNU Emacs *141*
7.2.1 Extensibility 142
7.2.2 Document Model and Typographic Display 145

7.3 Deep Extension vs. Scripting an API: Dynamic HTML, Microsoft Word, FrameMaker *146*
7.3.1 Dynamic HTML 146
7.3.2 Microsoft Word, FrameMaker 148

7.4 Client vs. Server *148*

7.5 Union vs. Confederation, or Build vs Reuse: Microcosm, UNIX Guide, Firefly *149*

7.6 Fundamental API vs. Open Source: Netscape 5, TkMan/Tk Text Widget *150*

7.7 Cross-format vs. Single Format: IDVI/TeX DVI *152*

8 Implementation Experience and Future Work **154**

8.1 Implementation Experience *154*

8.2 Future Work *155*
8.2.1 Requirements for Widespread Distribution 155
8.2.2 Second Generation Work 156
8.2.3 Research Issues 158

9 References **160**

Acknowledgements

Robert Wilensky's intellectual curiosity was open to pursuing an area of inquiry outside of his historical field of expertise. His patience and generosity sustained me through uncertain periods of progress. His ability to see the promise in the work and his vigorous advocacy to other respected figures in the field, always presented charismatically and with great technical accuracy, periodically reminded me that the old news to me was still good news. He was the primary audience on technical matters and the source of many technical goals.

Discussions with Gary Kopec proved invaluable, especially during the gestation period before and after quals. His Image EMACS is still a goal to achieve. Indirectly, the quality of the research at PARC has long been an inspiration.

Larry Rowe thoroughly reviewed the manuscript and made many insightful comments, particularly the addition of the incremental algorithms performance measurements section. Over the years he provided forums in which to present this and other work.

In Mary Kay Duggan's classes I gained an historical perspective on printing and publishing.

Mike Schiff's experience and generosity of time and expertise made for a better quals exam and Introduction. Carl Staelin carefully read and made invaluable comments on two complete drafts. Daniel Wilkerson's enthusiasm is contagious; and as someone who has written at least two Ph.D. Introductions himself, he led to a better one here.

Mario Silva suggested that there's this new thing called Oak (later called Java) that could be interesting.

Early adopters of the research software helped improve it. Wojciech Matusik implemented audio and video, and sparked significant improvements in architectural

generality. Loretta Willis reimplemented table sorting on scanned page images, server scripts to bridge to other servers and to save, and was an amazingly acute bug fixer. ByungHoon Kang wrapped James Clark's SP to regularize HTML from the web and fixed many bugs. Steven Rosenberg, Carl Staelin, Nic Lyons, Ben Vigoda, Shouyi Hsiao, Phil DeGreen, and Hansen Hsu at Hewlett-Packard have been doing interesting things since the architecture-free prototype. Brave souls in the Fall 1996 Digital Documents class extended the code in many interesting directions. Marcel Kornacker tied annotations to a database for his Internet Services class. Continuing interest from Owen McGrath, Howie Lam, the guys at the University of Essen, and Electra Sutton of the FBI kept motivation high.

Members of my former research group under Professor Michael Harrison, especially Wayne Christopher and Ethan Munson, explained the lore of CS grads.

I thank Adam Sah for many adventures.

Every Berkeley CS dissertation I have read profusely thanks Kathryn Crabtree, and I know why. She pierces the bureaucracy for you and is truly a Den Mother Extraordinaire.

Industry donated equipment on which the research was performed. Most recently of personal use, Sun Microsystems donated the UltraSPARC on which early development was done, as well as admission to several JavaOne shows. Hewlett-Packard donated the Wintel box of later development, a scanner and a printer.

Finally, I owe a great debt of gratitude to the governmental sponsoring agencies—NASA, ARPA, and NSF—who initiated a program in digital libraries at just the right time.

1 Introduction

No experience is more ubiquitous or omnipresent among computer workers than manipulating digital documents. Word processing, with its easy revision of documents, has replaced typewriters. E-mail, with effectively instantaneous delivery times and minimal cost distribution, has superceded paper to a large degree for many applications such as personal letters, memos and newsletters. More recently, web browsing has exploded in popularity, with both individual and corporation alike utilizing the network to publish at low cost and with wide distribution all manner of information. Other widely used applications of digital documents include news readers for Internet discussion groups, help browsers and text editors. And it is now widely economically feasible to maintain large collections of scanned-in images of paper documents, giving yet another type of digital document. More than any application of computing—spreadsheets, scientific computing, engineering simulation, multimedia design, even games—digital documents are crucially important to almost everyone, almost every day. The quality of digital document systems is a major factor in the value of computers, and therefore the improvement of such systems is worthy of investigation.

1.1 Problems: Documents Are Diverse and Not Adaptable or Progressive

1.1.1 Mediocrity through Diversity

One prominent characteristic of today's digital document systems is their great diversity. One source of diversity is a healthy survival-of-the-fittest competition among implementations. Another is historical, based on delivery mechanism, with complete,

independent separate systems for e-mail (which composes and browses messages sent directly to the recipient over the network), news reading (which composes and browses messages sent to discussion groups over the network), word processing (which composes documents usually kept locally, often for distribution through paper), and web browsing (which browses pages found at various sites on the network).

Another source of diversity is the role of the document. A help system should generally consume few resources and emphasize searching and navigation. A programmer's text editor should integrate with debuggers, parsers, program visualizers, project management, source code control, and other tools. As it is routed along a multi-stage path, a document in a workflow system must carry state and rules. Yet again, separate systems support each role.

A third source of diversity is the nature of document collections. Any large collection has unique characteristics, best examined with special tools. A collection of ancient manuscripts might contain different versions of essentially the same work. It might be stored primarily as scanned page images, and supported by tools to help the scholar transcribe, translate and compare the texts. Another collection might consist of governmental environmental plans posted for public comment. This collection could benefit from associated discussion threads, as well as discussion summarization by issue, relevant portion of the document, and author. Moreover, with the mass availability of corpora spurred by the web, demands on a document system can change dynamically and quickly. In general, a system should balance the preferences of its user with the needs of the particular document under consideration at that moment.

Yet another source of diversity is document data format. Some formats focus on appearance, including PostScript, PDF, DVI, and images of scanned paper; some emphasize semantic or logical markup, including SGML/XML and Texinfo; and others are combinations thereof, including ASCII, TeX/LaTeX, RTF, MIF, HTML and word processor binary formats. Again, distinct systems are built to support each type of document format (or vice-versa). This results in strong support for those formats favored by large companies (Microsoft Word binary format, Adobe PostScript), while others languish overall, surviving due to unique value to a niche audience (TeX's unmatched mathematics and linebreaking; Texinfo's suitability for both online browsing and book printing).

This great diversity results in serious problems for both user and software designer. The user must shift to a different system for each slight difference in document characteristic. Although local conventions often mitigate differences in user interface, the user nevertheless faces different software with different capabilities, each presumably the best of a large selection of competitors, but each lacking in different ways. Furthermore, each user works in an unique environment and has unique tastes and needs. The user could usefully personally tailor the document systems, not just the user interface but the capabilities of each along all phases of document work: writing, managing, distributing, reading.

Just as seriously, this diversity diffuses the energy of the software designer. Even if some useful feature, say full-text search, is successfully integrated into a system, that system operates on only a subset of the documents that could benefit. And that system is only one of the many for that particular purpose. One can argue that a principal reason the Multipurpose Internet Mail Extensions (MIME) did not rescue most e-mail from plain ASCII is that the effort of updating myriad mail readers proved insurmountable. Although MIME-capable mail readers exist, MIME is one feature among many, and one can see in retrospect that MIME itself is not compelling enough to entice users comfortable and knowledgeable with one system to abandon it.

The sources of diversity will remain. Documents will continue to be delivered in different ways, documents will continue to be used for widely different purposes in conjunction with other supporting tools, the number of document collections is only increasing, and different document formats will persist both for historical reasons and because in general one cannot be translated into another without loss of information (consider converting a scanned handwritten manuscript page image into SGML).

The splintering effect of this diversity on digital document systems, however, is of illegitimate birth. In different ways, Emacs and DeskScape [Brown 1995] demonstrate that the delivery mechanism is unimportant, that significant benefits accrue to integrating systems in a single framework. The Ensemble project [Graham *et alia* 1992] studied the usefulness of unifying general document services with programming language support. Java applets partially address the varying needs of document collections, but suffer from coarse integration—as islands of interactivity scattered in the document proper. One aspect of this thesis defines a framework that abstracts the differences of different document formats.

1.1.2 Documents and Systems Not Adaptable or Progressive toward Future

By now, complete systems expected by users are large and complex. The software designer with a good idea faces the choice of trying to integrate it into existing software or to build the rest of a system to support the idea.

Unfortunately, existing systems are not generally amenable to new ideas. While many do expose their document services for external control, seamless integration of many ideas requires the document services themselves be extended. No previous system was sufficiently flexible to extend this power. Related work is examined in depth in a later chapter, but briefly consider three popular systems that seem to offer such extensibility, but in fact come up short in critical ways. FrameMaker, Microsoft Word, and other systems offer application programming interfaces (APIs) by which their document services can be manipulated by user-written programs. As a rule, APIs protect their internal data structures, providing a higher-level semantic interface to the programmer. This is sufficient if the software designer's idea can be implemented with a combination of existing features. Yet, if it truly is a *new* idea, it is unlikely that the API can accommodate an innovation outside of its original conception. In short, APIs provide a

programmatic means to invoke existing functionality, but do not allow changes or improvements to the functionality itself: it is interfacing, not extension.

HTML seems to offer arbitrary extension with Java applets or native code plug-ins, which can be authored in a general-purpose programming language and easily added to documents. Unfortunately, this extensibility is not finely integrated with the document proper. Applets are simply rectangular islands of extension in the ocean of unextensible HTML. While applets can be used to implement mathematical typesetting or experiment with a new type of hyperlink, they are clearly foreign objects unresponsive to linebreaking and the current font. For extensions that naturally would manipulate more of the document than a small rectangle, the limitations of applets are starkly apparent. For example, a commercially available speed reading applet [Tenax], which flashes each word in the page one at a time, must first be loaded with a duplicate copy of the page text processed to remove HTML markup, and knows no better than to start at the first word and march linearly to the last. A better design would use the existing page text and allow the user to browse the page, invoking the speed reading function on desired passages, but this is not possible with an applet. The type of extension exemplified by HTML and Java applets is not integrated with the document so much as juxtaposed with built-in features and other extensions.

In the foreseeable future Dynamic HTML will provide much greater control over and access to HTML. Yet, assuming it is possible to integrate such new capabilities with a good user interface, this still leaves unimproved all the other types of digital documents one deals with: e-mail, program editors, and so on.

Recently Netscape Communications released the source code to their Navigator browser for public consumption. This would seem to solve many problems. Aside from the fact that a software designer has more than a million lines of program code to master, any new idea can be realized with enough hacking, with the benefit that most of the rest of the system already exists. However, this neglects the necessity of a synchronization phase, where the work of many developers is unified and the improved version becomes the new common base. So-called “open source” development with the participation of many developers can work well when the goal is well defined, such as rewriting an existing operating system like UNIX or duplicating UNIX utilities. It works less well for new ideas that need maturing or that appeal to a niche audience, and hence are likely to be rejected on the valid reasons that the unified result should be stable and not so large as to become unmaintainable. Distributing innovations independent of the main thrust of development does not scale, as changes by different authors are likely to conflict irreconcilably, sooner or later.

The other alternative to spreading innovation—building an entire new system—just to support the new idea (or desired missing feature) is an obviously unfortunate duplication of effort, but frequently necessary. The lack of hyperlinks in the UNIX help browser xman spurred the creation of TkMan, which had to duplicate nearly all of xman’s features to be competitive.

More often, the enormity of the task overwhelms the lone inventor into being satisfied with a small, experimental implementation good enough for personal use and possibly an academic paper, but unsuitable for public distribution. Unless it is revived by a monolith owner, the idea dies—ultimately disappointing a potentially large audience of users. Just within the last few years did Microsoft decide that the concept of hyperlinks was a good idea and support it within their word processor and other products; chances are that your new idea is further down their implementation list.

The situation is analogous to that which spawned the scripting language Tcl. Prior to Tcl, a new scripting language was built largely from scratch for many large software systems. Since the extension language was not the focus of any of the projects, little effort was directed into their corresponding scripting languages, with predictable consequences. A focus on the scripting language itself produced Tcl, a robust language for scripting that could be shared by and benefit many authors and their users.

Just as the great diversity of digital document systems diffuses the energy of the software designer, the lack of deep extension that is finely integrated and easily distributable, which is present in some combination in these systems, stifles innovation.

1.2 Goal: Extremely Improvable Digital Documents and Systems

Taming the pernicious effects of diversity is half the battle; it solidifies the groundwork. The other half of the battle is to build on that groundwork, realized as a software framework, as a launching pad for new and useful innovations. The challenge is to give the software designer with an idea the *power* to extend as well as control existing document services by defining the abstractions that give the framework the *flexibility* to accommodate unanticipated experimentation or innovation. Furthermore, given numerous ideas simultaneously competing for attention and resources, the framework must *compose* these extensions to produce a coherent whole at no disadvantage to monolithic systems conceived all of piece.

If the battle can be won, resulting in a new, radically open environment, the individual software designer can take advantage of the existing functionality in the framework to stop wasting time duplicating work and concentrate on innovation. Rather than die for lack of a supporting system, innovations can be distributed easily and quickly have a real world impact. Users can choose among fine-grained extensions, rather than large take-it-or-leave-it bundles, to compose a highly tailored system. Innovations can have narrow appeal; since the amount of implementation work is roughly proportional to the innovation, designers can satisfy niches not large enough to catch the attention of today's large, horizontal applications. Given common base abstractions over the diversity, innovations to the base feed back into and improve all constituents.

Phrased as a slogan, the goal is “anytime, anywhere, any type, every way user-improvable digital documents and systems”.

Anytime. It is a common experience to visit numerous web sites in quick succession. Many sites could usefully provide special tools to examine their content, and conversely the user may want to selectively apply tools, such as filters and visualizers. The system should augment its capabilities on demand. Likewise, content should be extensible at any point in time. Annotation is a common, if usually superficial, way of adding to content. It should also be possible to improve the content of the original document, say correcting the OCR translation of a scanned paper document, on demand by the individual with a strong interest in that document, and for the augmented results to be the base for further document manipulations.

Anywhere. With the great popularity of e-mail and the web, good support of networked communications is important, without neglecting the local file system. Rather than first requiring all web sites to update with some new feature in order to cooperate, improvements should operate in existing environments. Some environments may be hostile, as for instance read-only media such as CD-ROMs or third party web sites are not amenable to storing annotations.

Any Type. As discussed above, many document formats (scanned page images, HTML) and genres (ASCII files given special treatment as e-mail) are popular. It is important that the system be adaptable to them. Software designers must be able to write some feature once and have it operate on all document types the system supports, or otherwise the system fractures along document type, reverting back to the status quo.

Every Way. A feature could potentially need to change any aspect of a system: content, functionality, or overall operation. All three aspects must be open deeply and flexibly. Some base core system is needed as a catalyst to bring together the otherwise improvable components, but this fixed core should be as small as possible. If such as system can be made to work, it is likely to prove quite versatile, as nothing significant is unavailable to extension.

User. The end user should be able to enhance and customize the system to taste. The software designer should be able to distribute features easily and to improve the system independently of other authors, which is to say the system must compose features and adjudicate conflicts.

Improvable. Some document and media formats, such as ASCII, are not very sophisticated but still very popular. It should be possible to add modern capabilities to these formats. From a perspective far enough in the future, as technology develops, all formats are inexpressive. All improvements should integrate seamlessly, and feel like a usable, well designed system.

Digital Documents and Systems. Just as Java was informed by the past twenty years of programming language research, any new digital document system should take advantage

of what have emerged as widely agreed upon principles, such as logical or structure-based document construction, structure-based formatting with style sheets, structure markup expressed as SGML or more recently XML, multimedia data capability, a graphical user interface, what-you-see-is-what-you-get (WYSIWYG) editing, built-in networking, and incremental algorithms to give good performance and scalability, among other best practices.

1.3 A Solution: Pervasively Open, Highly Factored Documents and Systems

This thesis confronts the great diversity and unadaptability of digital documents and digital document systems. The fundamental architectural approach, called the *Multivalent Document Model*, (1) highly factors documents into components of content called *layers*, (2) highly factors their supporting systems into components of functionality called *behaviors*, and (3) defines the basic document operation as a set of protocols, which buttressed with additional mechanisms, allows these components, often authored without specific awareness of one another, to compose as a seamless unity.

Given the diversity of document data formats, the mix of multimedia formats into the standard text and image formats, the potential for non-traditional data formats such as geographic information systems or computer-aided designs, the necessity of bringing expected modern functionality to diverse formats (hyperlinks to scanned page images), the need to associate metadata and other ad hoc information with documents, and unknown but expected other pressures in the future, the definition of a single new all-encompassing data format seems infeasible (or at least its implementation would seem so). In its place is proposed a conceptual approach to building documents out of often heterogeneous, but always coherently related layers of content.

Correspondingly, the supporting document manipulation system is built piecewise, out of behaviors. No behavior is built into the system. All behaviors are loaded dynamically, on demand, instantiated and coordinated by a small core. Some behaviors supply general system functionality such as searching. Others mask some kind of document diversity such as document data format, presenting a uniform abstraction to other behaviors. Except for an anticipated relatively small number of behaviors specifically tasked to mediate between concrete document details and the abstractions, all behaviors operate on the abstract document model, and therefore their services are available to the diversity of documents without special consideration by the behavior for any one. In fact, all user-visible functionality is supplied by behaviors.

Such highly factored content and functionality embrace the widest diversity. After all, behaviors are more or less code and layers are data, and code and data are the components of all computer programs. The challenge addressed in this thesis is to devise

a mechanism of composition to bind layers and behaviors into conceptual wholes. The Multivalent Document model defines abstractions for documents—a framework within which the highly factored components can be placed. Sophistication arises from composition of simple, heterogeneous components. At a high level the system should feel to the user as if it were custom built exactly for the application at hand. At the low level, the software designer's behaviors are given great power to control all aspects of the document, which could easily result in collisions between behaviors competing for the same resources, especially considering the fact that behaviors can be written by different authors at different times. A profitable tension between freedom and cooperation is achieved with a combination of overall control by the core system and adherence to well defined protocols by behaviors. As demonstrated below, the architecture has proven equal to a diverse set of applications, yet it is not biased toward or specially accommodate any particular one, which gives hope that it might be equal to future demands not specifically anticipated at present.

The system that realizes these concepts was developed as a project of the Berkeley Digital Library Project [Wilensky 1996]. It was implemented in Java and has been deployed at the Project's web site (http://elib.cs.berkeley.edu/). One reason for the name owes to the Merriam-Webster Dictionary's definition of "multivalent" as "having many values, meanings, or appeals", relating to the multifarious benefits to users, software designers, and corpora managers. The other derives from an atomic analogy. Atoms bond with one another by sharing their outermost, or *valence*, electrons, forming molecules with useful emergent properties. Likewise, behaviors and layers unite to form single conceptual documents with powerful emergent capability from simple parts, but with more extensive sharing and interaction among components.

The following section exhibits the proof-of-concept Multivalent Document system as applied to three applications: enlivening scanned page images, extensible HTML, and distributed annotations in situ, pointing out the layers and behaviors involved. The section thereafter summarizes the architecture of the system. Although the architecture may appear simple (it is hoped), in fact it sufficed for and contributed significantly to the realization of these diverse applications. No special accommodation was made in the architecture for any of the three applications.

1.3.1 Demonstration

Enlivening Scanned Page Images

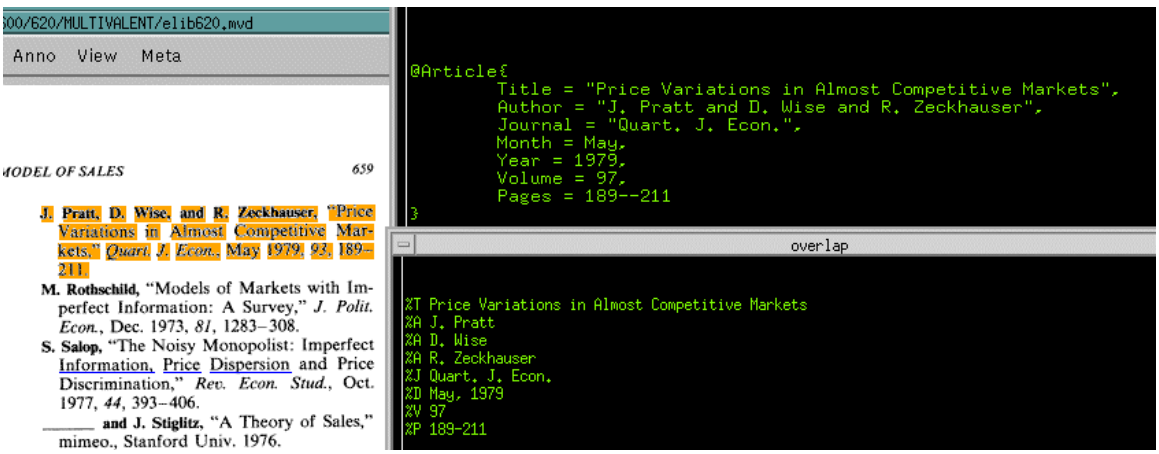
First consider the document obtained by scanning a paper document into the computer. One can run optical character recognition (OCR) on the image to generate an approximate translation as textual characters (ASCII). Each format has its advantages. The image best captures the original paper document, with its illustrations, fonts and formatting. For the reader, that is; to the computer it is little different than a picture of a sunset. The OCR translation can be searched and edited by machine, yet for all but the simplest of documents, some degree of error was introduced, which may or may not have

been acceptable. For many applications, including historically significant and legal documents, any degree of information loss is unacceptable.

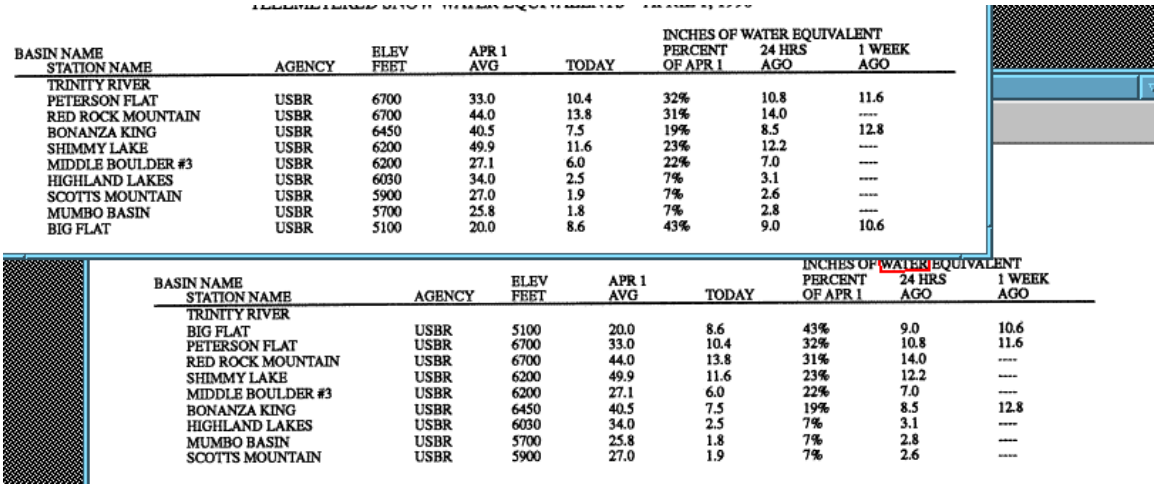
One application of the Multivalent system treats the scanned image and OCR text as layers and unites them with a third layer that geometrically maps between them. The application then displays the high fidelity image yet enables document manipulation based on the OCR. The model gains power by semantically aligning layers in this way.

As illustrated in the screen dump below, the OCR was searched for the words "water" and "association" and the matches boxed in the image. And a selection was made in the image starting at "District" in the leftmost column and dragging to "Department" in the rightmost. The text of the selection is placed in the system cut buffer where it can be pasted into other applications. Various layers are best suited for different applications; by aligning layers, the most amenable representation can be utilized, and the results reflected back to the others.

This specific functionality has been duplicated (by Adobe's Capture and Acrobat), but working in the Multivalent model one can add additional layers of information content, which behaviors can utilize to produce additional useful functionality. For instance, anticipating technology for advanced document analysis (which has been simulated here manually), the bibliographic entries in the screen dump below have associated with them, as a layer, their semantic categories: author, title, pages, publisher, among others. A behavior weaves this information with the other layers. Other behaviors can intercept the copy and paste operation on the bibliographic regions and compute on the fly alternative representations more useful to the task at hand, such as the BibTeX and refer formats shown at right. Other concrete bibliographic translations can be added by writing program code that simply formats strings as desired. A similar translation has been done for math, where the selected formula can be pasted from a pre-generated set of translations (OCR, Lisp, TeX).



In another case, advanced document analysis (again anticipated) has described a table in the document as supplied as another layer. A table sorting behavior uses this information to take mouse clicks on column headers in the table and sort the table based on the corresponding OCR, displaying the results by correspondingly editing the document. In the screen dump below, the user clicked in the upper table on the column labeled "PERCENT OF APR 1" and this behavior sorted numerically to achieve the result in the lower table. This behavior happens to determine the data type of the column, alphabetic or numeric, and sorts appropriately, and can sort in reverse with a shift-click.

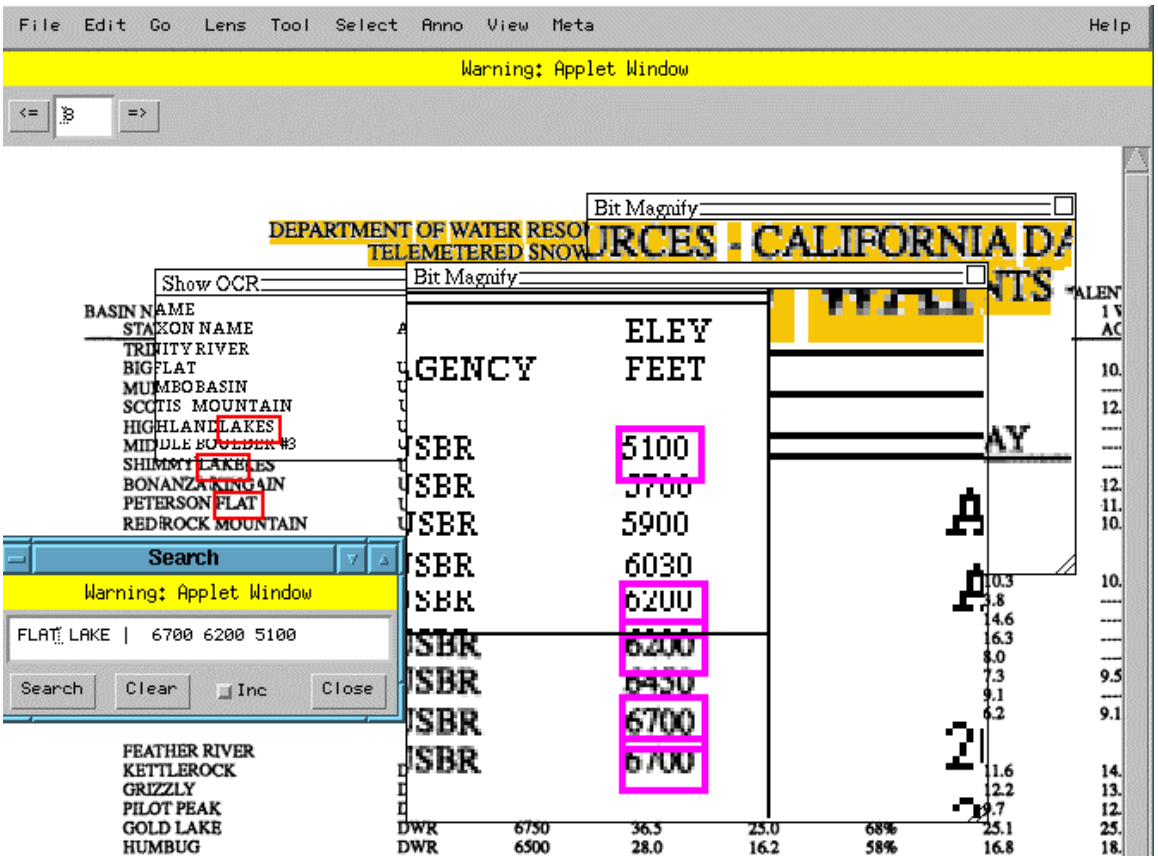


BASIN NAME / STATION NAME	AGENCY	ELEV FEET	APR 1 AVG	TODAY	INCHES OF WATER EQUIVALENT PERCENT OF APR 1	24 HRS AGO	1 WEEK AGO
TRINITY RIVER							
PETERSON FLAT	USBR	6700	33.0	10.4	32%	10.8	11.6
RED ROCK MOUNTAIN	USBR	6700	44.0	13.8	31%	14.0	----
BONANZA KING	USBR	6450	40.5	7.5	19%	8.5	12.8
SHIMMY LAKE	USBR	6200	49.9	11.6	23%	12.2	----
MIDDLE BOULDER #3	USBR	6200	27.1	6.0	22%	7.0	----
HIGHLAND LAKES	USBR	6030	34.0	2.5	7%	3.1	----
SCOTTS MOUNTAIN	USBR	5900	27.0	1.9	7%	2.6	----
MUMBO BASIN	USBR	5700	25.8	1.8	7%	2.8	----
BIG FLAT	USBR	5100	20.0	8.6	43%	9.0	10.6

BASIN NAME / STATION NAME	AGENCY	ELEV FEET	APR 1 AVG	TODAY	INCHES OF WATER EQUIVALENT PERCENT OF APR 1	24 HRS AGO	1 WEEK AGO
TRINITY RIVER							
BIG FLAT	USBR	5100	20.0	8.6	43%	9.0	10.6
PETERSON FLAT	USBR	6700	33.0	10.4	32%	10.8	11.6
RED ROCK MOUNTAIN	USBR	6700	44.0	13.8	31%	14.0	----
SHIMMY LAKE	USBR	6200	49.9	11.6	23%	12.2	----
MIDDLE BOULDER #3	USBR	6200	27.1	6.0	22%	7.0	----
BONANZA KING	USBR	6450	40.5	7.5	19%	8.5	12.8
HIGHLAND LAKES	USBR	6030	34.0	2.5	7%	3.1	----
MUMBO BASIN	USBR	5700	25.8	1.8	7%	2.8	----
SCOTTS MOUNTAIN	USBR	5900	27.0	1.9	7%	2.6	----

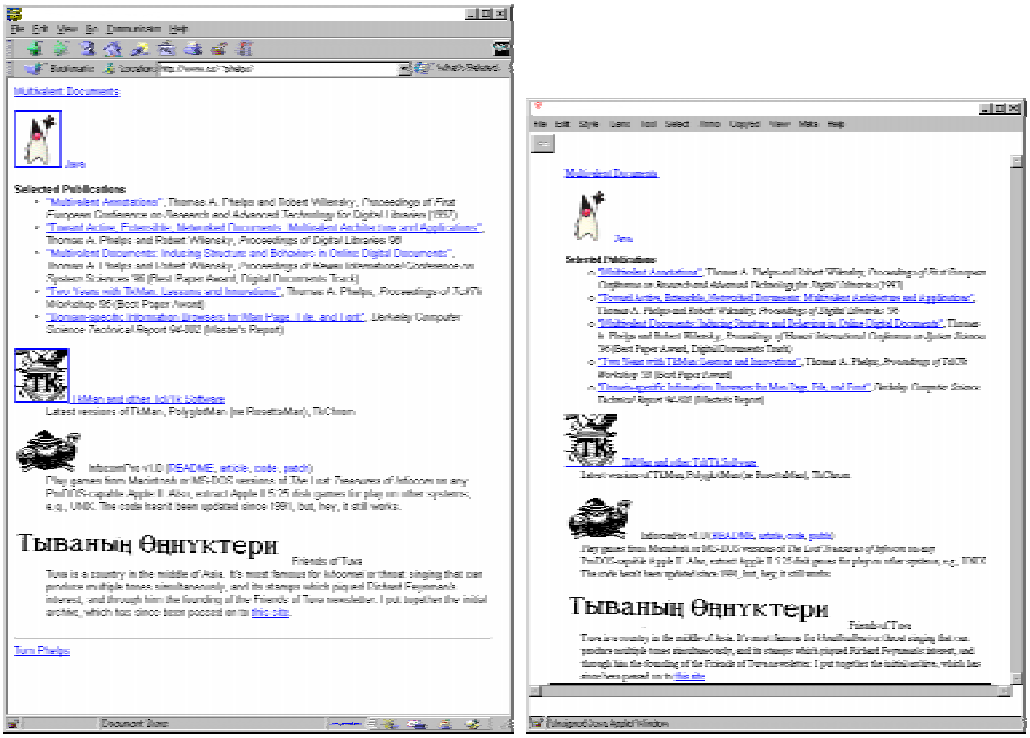
A type of behavior called a lens, implemented without special consideration in the architecture, can arbitrarily transform the contents of a rectangular window on the screen. Because behaviors have full access to the document content, transformations can be more sophisticated than simple pixel processing. Moreover, working within the prescribed Multivalent protocols, lenses compose with each other and with other behaviors without further effort by the behavior author. The screen dump below exhibits a "quintuple backflip", that is, five behaviors interoperating. An OCR lens shows the OCR translation for the corresponding region of the image. A magnifying lens doubles the size horizontally and vertically (quadruples the area) of the region; where it overlaps the OCR lens it displays magnified OCR, otherwise it displays magnified bit image. A second

magnify lens doubles the first where they overlap, resulting in overall quadruple magnification. The lenses compose with a selection, a sorted table, and search hits.



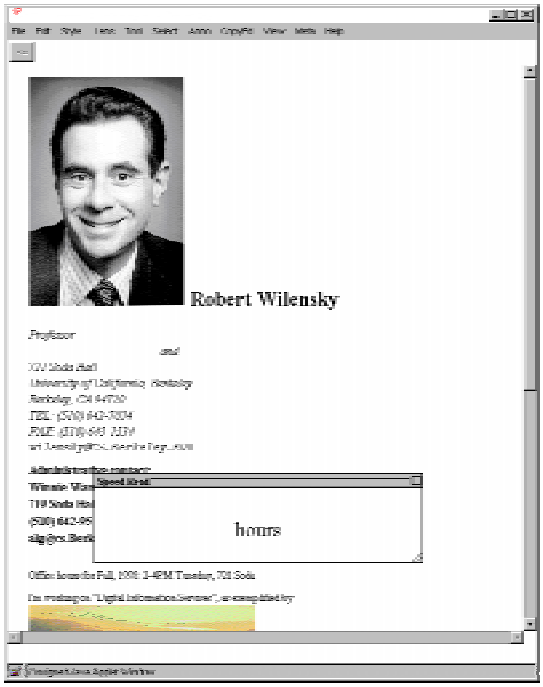
Extensible HTML

Below at left is the author's HTML home page displayed in Netscape. Below right is that HTML displayed in the Multivalent system. As a document format, HTML differs to the extreme from scanned images. Images begin with appearance only, and what semantic content is known is derived through document analysis. HTML begins as content with (ideally) semantic markup, which the system formats and displays as an end result. HTML in the Multivalent model can be usefully extended with additional tags as the author can associate behaviors with new tags and distribute the behaviors with the document, retaining the document's broad accessibility. Publishers could use this extensibility to duplicate the layout control beyond that available in HTML that they are accustomed to for their print publications, while preserving the semantic structure of the document for the end reader, who, in turn, may want to semantically manipulate the document contents, as by sorting the tables differently. As another illustration, mathematics has layout and semantic components with strong manipulation potential, and the great variety and continuous growth of mathematics makes it unlikely that a fixed tag set that can evolve only through approval by standards committees will adapt swiftly enough to meet all needs.



A behavior shown below at left has defined the highlighted word. This capability itself is commonplace; what is interesting about this implementation is that, rather than relying on a tightly bundled dictionary, the behavior utilizes a network service that happens to be available locally. In other words, the behavior is custom integrated to the unique local facilities. As another example, a local text-to-speech system could hypothetically be usefully employed, benefiting sight-impaired users and anyone else whose visual attention could be better directed elsewhere. Moreover, such site-specific services may be superior to bundled facilities or may link to services, such as web search engines, that can only exist on the network. And because of the Multivalent model, text vocalization and these other new services are available to all document types.

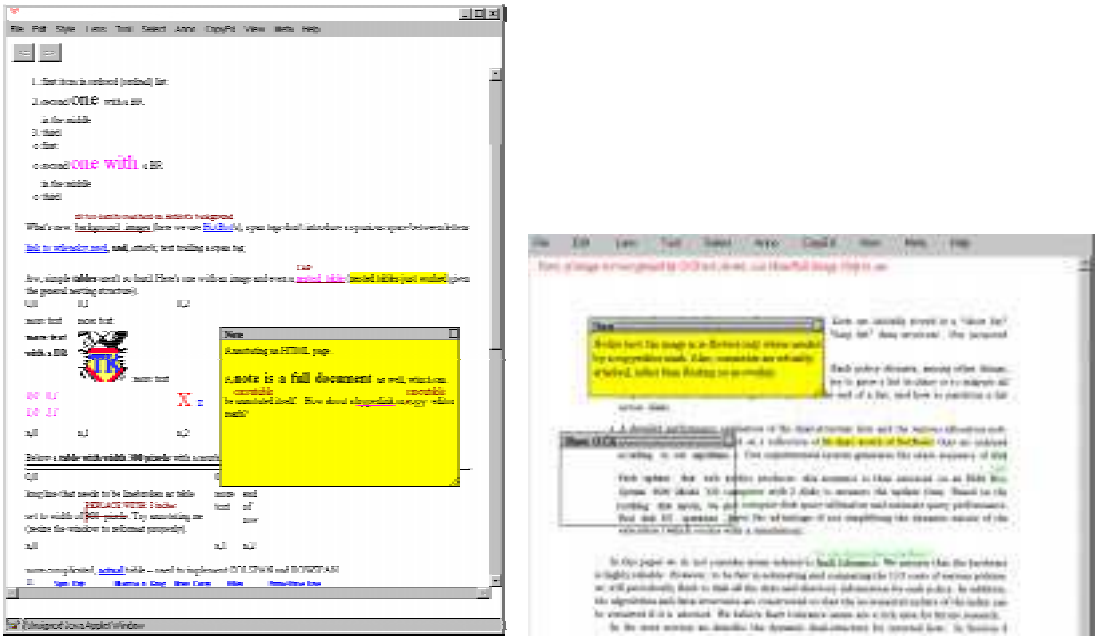
The screen dump at right displays the speed reading behavior whose implementation as an applet was mentioned as an example of applet limitations. The behavior implementation has access to the document content, rather than requiring the applet's duplicate copy; and can be started at an arbitrary point after gross browsing of the page (beginning at the cursor or selected word), in contrast to the applet's sequential march from first word to last. Moreover, the behavior implementation operates without modification over different document types.



Distributed Annotations In Situ

Both scanned images and HTML can be annotated in situ. Below at left an HTML page is annotated with a highlight on “nested tables just worked”, a variety of copy editor markup, and a note, which is itself annotated. In fact, some of these copy editor marks are executable; clicking on the “REPLACE WITH” or “CAP” marks executes the intention of the mark, materially replacing and capitalizing the text respectively. At right a scanned page image has been annotated. Notice that, although the formatting is fixed, the document was subtly reformatted to open up space for annotation text between lines. The same behaviors that annotate HTML can do so as well for scanned page images, since behaviors manipulate an abstracted document tree, not the particulars of the medium itself. In other words, annotation functionality has been separated from media-specific manipulation. Concrete knowledge of a medium, such as how to paint its content, is encapsulated in a behavior type called a media adaptor.

Annotations are stored apart from the original document so that they can be added to documents formats that do not support annotations themselves and so that read-only documents (on a foreign server or CD-ROM) can be annotated without making a copy of the original document. Annotations are robustly anchored, meaning that if the annotated document changes annotations can still usually be reattached; reattachment capability degrades gracefully with increasingly large changes to the original. Annotation makes natural use of the layer metaphor, where sets of annotations by a single author can be collected as a layer, and simultaneous commentary by multiple authors is supported by composing the corresponding layers.



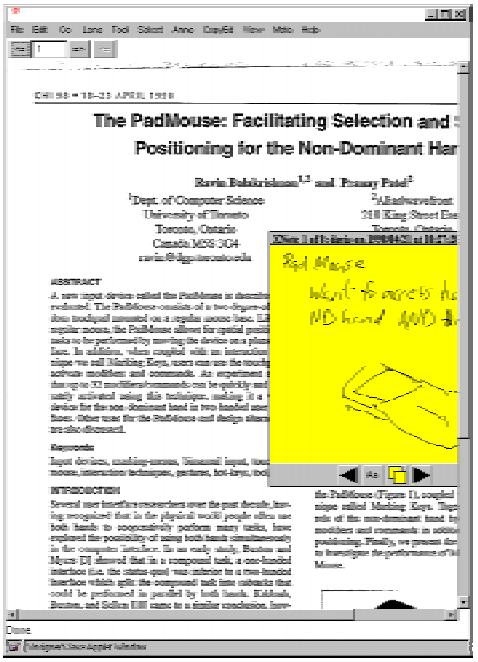
A novel annotation type is demonstrated below. Below at left, an HTML page has been annotated. The user chooses the menu item "Executive Summary" to produce the screen dump at right. The document collapses to show all annotations, in both original document and note, along with local context and structural context (H1, H2, … tags). Clicking in the collapsed document expands it and scrolls to the corresponding point. This annotation type is named Notemark, after its dual role as a note providing localized information, and as a bookmark which can lead to the full text. This idea translates well to a different genre, to UNIX manual pages, that operating system's online documentation, where Notemarks can be usefully made for highlights, excerpted first lines of paragraphs, search terms, "autosearch" hits, subcommands, and command line options. This shows that the Notemarks property composes fluently with other behaviors.

Third-party Applications

Although the system as yet lacks the feature set and robustness to replace one's current document systems and lacks documentation for software designers, the system has

enabled a number of projects by third parties. The screen dump below left shows a video playing with, in the two note windows, synchronized subtitles in English and Polish [Matusik 1998]. That application can synchronize arbitrary features such as hyperlinks, highlights, and the manifying lens with time-based media including audio and video.

The screen dump at right composes a scanned conference paper (from the Association for Computing Machinery's digital library) with handwritten notes, taken on a PalmPilot device during the conference talk on that paper [Li 1998]. Each note is a full-featured document in its own right, and can be viewed as recognized ASCII or annotated itself. Other current work by third parties include application to foreign language teaching (with a Chinese language translation lens) [McGrath 1996] and in the research for a Ph.D. thesis entitled "Linking Past and Future; an Application of Next Generation Computing Technology for Medieval Manuscript Editions" [Sanderson 1998].



1.3.2 Architecture

To recapitulate, in the Multivalent model, documents and their corresponding systems are highly factored into layers of content similar to program data and behaviors that provide functionality, and systems are built from the composition of heterogeneous sets of layers and behaviors. Few restrictions are made on components, thus admitting the widest flexibility to embrace current and future technology.

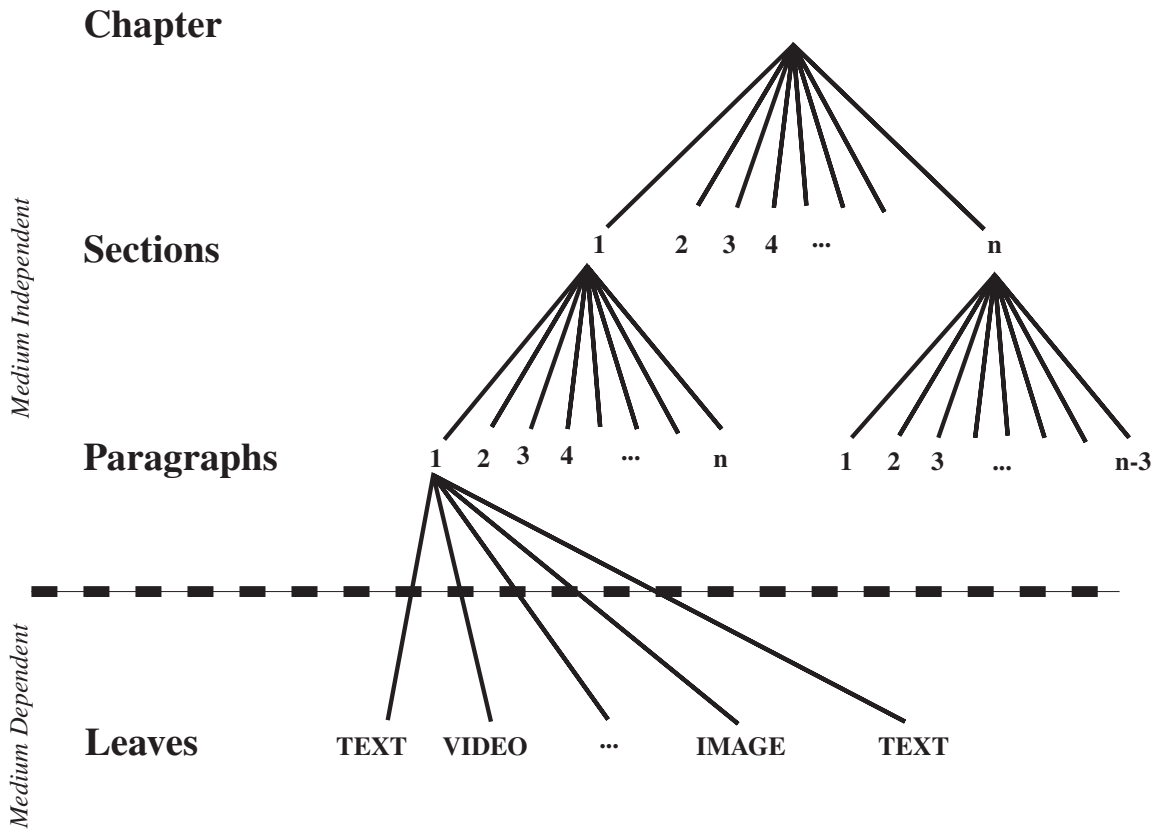
The challenge for the architecture is to provide a general document model so that (1) behaviors can be generally useful, operating over many document data types and media formats, without concern for media-specific details, (2) behaviors are given the power and flexibility to access and potentially filter all document content and operations on

documents, and (3) behaviors compose, summing their effects when they do not conflict and orderly adjudicating when they do. (Since layers are data, which are inert in themselves, the challenges are given in terms of behaviors, some types of which can be seen as the runtime embodiment of corresponding layers.)

Beyond this, the supporting architecture embodies modern ideas of document construction, with a logical document tree, a WYSIWYG document view and graphical user interface, spans and lenses, incremental algorithms that free CPU cycles for more complex content, robust locations for the constantly changing networked world, among others. Three applications—enlivening scanned page images, extensible HTML, and in situ annotations—validate the basic architectural concepts and suggest an interesting future as the Multivalent Documents System moves into more public availability.

Document Tree

Behaviors operate across layer data types by manipulating the abstract document definition embodied in the document tree. The document tree is a tree data structure that reflects the logical structure of the document, to the extent possible for that data type. Internal nodes of the tree reflect the logical hierarchy of the document. For instance, a book comprises chapters, which comprise sections, which comprise subsections, which comprise tables and paragraphs; tables comprise table cells, which comprise paragraphs. Paragraphs may comprise text, graphics, or other media types. Other media types can usefully augment the overall structural representation of the document. A video type could structure its content in the tree as scenes, comprising sequences, comprising shots, comprising frames, comprising (with computer vision analysis) figures.



At the leaves of the document tree can be found the medium-specific elements: words, graphics shapes, frames or figures. A behavior known as a media adaptor encapsulates knowledge of a layer and integrates its content into the document tree, both structurally and at the leaves. All access to the corresponding layer is interpreted by the media adaptor, so that other behaviors can operate on the abstract document tree, and requests to leaves, say to draw the content with a given background color, will be carried out by the media adaptor without burdening general behaviors with media-specific knowledge. (Any scheme that builds in special case knowledge for individual media types obviously could not meet the stated goals of accommodating future innovations.) This method of exposing the structure of data for manipulation by an extensible set of components could be generalized to different domains; this dissertation describes it application to the domain of digital documents.

Protocols

How can an architecture claim to support experimentation and unanticipated new ideas? The Multivalent architecture for digital documents takes all phases of what we term the fundamental document lifecycle, following a document from persistent storage to runtime manipulation and back, and lays them bare, as operational protocols, for manipulation and mutual filtering by arbitrary behaviors.

To some degree all digital document systems implement a number of phases as part of the fundamental document lifecycle: reading the document from persistent storage,

building the internal representation, formatting content, painting the internal content on the screen, receiving the user's instructions via mouse and keyboard and other input device events, cut and paste or drag and drop interchange with other applications, undo/redo, and saving. In other systems, various code can be categorized as mainly involved in one phase or another, but in the Multivalent model, each phase is reified—treated as a concrete protocol—and opened to control as requested by interested behaviors with well defined information passing between core framework and behaviors.

A behavior is a piece of code that implements these protocols. All user-visible functionality is implemented by behaviors. The core framework instantiates behaviors and passes control among them. One behavior can modify the result of another or even prevent another behavior from being invoked, by requesting to be invoked at the appropriate time and taking the appropriate action then. These two ways to filter other behaviors correspond to the way each protocol is divided into two *subprotocols, "after" and "before"*. For instance, when one behavior paints after another, it can draw on top of the other's result, but if it draws before, it can "short-circuit" the other, and the other is not given a chance to paint.

In addition, behaviors have a hook to the document tree and a set of navigation, inspection and manipulation operations that together give full access to the document content.

Together, a behavior's integration with the fundamental document protocols and its access to full document content are powerful enough to support behaviors that clearly have not been specifically accommodated in the framework. For example, the OCR lens shown in the previous section is implemented as an ordinary behavior, at no disadvantage in performance to an implementation that were to be coded in the core framework. By opening the fundamental level to extension, some high level functionality may be more laborious to implement, but everything should be possible. The power of fundamental protocols coupled with full content access suggests that the Multivalent framework can support innovations in document technology, for what other control or access could be needed: there are no other functionality or content, analogously to the quotation below:

> Anyone who wonders about the accuracy of [professor of business administration Julian L.] Simon's data or conjures up rafts of competing data on the other side of the issue will be met with Simon's claim that: "There are no other data." His statistics, he claims, come from the "official" sources, the standard reference works that everyone uses.
>
> "The Doom Slayer", Ed Regis, Wired 5.02

Protocols interact with the document tree variously depending on the protocol. The document lifecycle for a stored, native Multivalent document begins with the system reading from a *hub document* the list of behaviors that comprise the document. The system instantiates the named behavior and invokes its *Restore* protocol method, at which time the behavior performs general initialization and reads the corresponding layer or layers it encapsulates, if any, either from data inline in the hub or, given the location of the layer, out of band. Next the system iterates over the behaviors so instantiated in the

Build protocol. First it executes the Build Before subprotocol for each behavior in order, then the Build After in the reverse order. During Build Before, behaviors build up the document tree. Behaviors called media adaptors bridge concrete document formats into the logical tree. In the Build After subprotocol, those behaviors that align their content on the basic tree, such as annotations, re-establish their *locations* (which are robust to changes in the base tree). If a behavior intends to modify the ordinary workings of a portion of the document, it *registers interest* in the appropriate tree node, either a leaf or internal (structural) tree node; thereafter, every time any protocol involves that portion of the tree, the behavior will be invoked with the capability of modifying the protocol's effects over that subtree to arbitrary degree.

With the document tree built, most remaining protocols invoke behaviors along tree walks. The logical structure of a document is assigned geometric positions for rendering the document appearance during the *Format* protocol. Format proceeds along a depth-first tree walk from the document root. Internal nodes first recurse to their children to learn their space requirements, then place them at (x,y) coordinates according to the layout algorithm. Likewise, the *Paint* protocol walks the tree starting at the root, painting the content of the tree, as placed during Format, on the screen or printer. Recall that media types are encapsulated at the tree leaves, and their corresponding media adaptors report the space requirements and can paint the contents of the media content at the exposed level of granularity.

To format and display document content properly, participants in the protocols must take into consideration a variety of graphical properties, including fonts, colors, and underlining. Rather than encoding these properties on each piece of content to which they apply, properties are bundled as a *graphics context*, which is carried along during tree walks. As the tree walk enters subtrees, the system checks the document's associated *style sheet* for structure-based display properties (perhaps 14-point bold text for section titles, hyperlinks underlined and in blue). Non-structural regions or *spans*, beginning at a point in one leaf and running continuously to another, likewise set properties in the graphics context. Conflicts among multiple entities over the same property are decided by the entities' self-declared priorities. Media types are obligated to respect the graphics context as much as possible for that medium.

With the document displayed, the system waits for user action in the form of events from keyboard, mouse or other input device. The *Event* protocol dispatches the event from the document root. Most often events reach the leaves of the document tree, where they are handled by the default event handler that manages the selection. Some behaviors *register interest in the root* to receive the event before it is passed to the document tree proper. Lenses use this to intercept mouse events that move and resize them, and the Magnify lens adjust the coordinates in the event to its distorted coordinate space. Within the document tree, the table sort behavior intercepts events on column titles to initiate sorts.

Clipboard and *Undo/Redo* protocols cooperate with individual behaviors to realize unified functionality across a disparate, extensible set of data types and operations. The Clipboard protocol, which includes cut-and-paste and drag-and-drop, iterates over the

portion of the tree described by the selection, if any. As Clipboard steps along the selection building up its content, it finds the largest subtrees entirely enclosed in the selection and performs a depth-first tree walk on each. For example, the Biblio behavior, which translates bibliographic entries to an extensible set of concrete formats on the fly, takes advantage of this by registering interest at the root of the subtree corresponding to an entry, computing the concrete format during the Clipboard Before subprotocol, then short-circuiting the remainder of the tree walk for that subtree. As with Format and Paint, Clipboard asks a leaf's media adaptor to generate the suitable representation of its content. Likewise, Undo/Redo treats behaviors as encapsulated black boxes, receiving for each action a behavior takes a "cookie" that can be presented back to that behavior to undo the action; Undo itself merely maintains a sequence of behaviors.

Means of Composition

Considering the power given to behaviors and the fact that, if the system gains popularity, most will be written without specific awareness of one another (even if mutual awareness were possible, it would result in a combinatorial headache), the potential for conflict is great. Thus, a primary aim of this thesis is to develop means of composition that harmonize behaviors. To wit, the following means have been defined and investigates:

Protocol definition and behavior adherence
Before and After subprotocols
Side effect through document tree
Behavior priorities
Global and graphics context attributes
Manager behaviors

The primary means of composition are the programmatic definitions and described intentional roles of the protocols and document tree, the API to which behaviors and, through them, layers adhere. Protocols invoke behaviors at appropriate times, at which point behaviors take action suitable to the protocol. Loosely coupled behaviors communicate through the document tree, which is at all times the definition of the document; if one behavior modifies the tree and another does after that, they perforce compose because they share the same data structure. Behaviors mutually manipulate one another during subphases of protocols by, generally, creating work during the Before subphase and modifying the results of other behaviors during the After subphase, with a possible short-circuit of lower priority behaviors or an entire corresponding subtree traversal. Although it is possible to abuse this power and write malicious behaviors, behaviors achieve a large degree of coordination simply by being written with knowledge of general protocol operation, hooking in at the right point and contributing their action to the whole.

Conflicts among behaviors are adjudicated by two systems of priority. An implicit priority is assigned to every behavior as it is incorporated into the running system. Those protocols that iterate over all behaviors do so in priority order, highest to lowest for the Before subphase, and lowest to highest for After. Protocols that make tree walks follow

this sequencing for all behaviors that have registered interested at tree nodes. A second, self-declared priority is used for behaviors that manipulate the graphics context, in which, generally, lenses override spans which override style sheets.

At times, groups of behaviors need to coordinate more closely. Sometimes it is enough to store attribute name-value pairs where closely coordinated behaviors know where to look, often the global namespace or in a special field of the graphics context. For instance, the current page number of multi-page documents is kept globally, where it coordinates the various document components, content, per-page behaviors, annotations, and so on; and the flag that determines whether a scanned page is drawn as image or OCR is kept in the graphics context where it is examined by each image-OCR leaf before drawing. The attribute's value can be more than a simple string or number. In the case of the user interface, it is the root of the abstract user interface tree; each behavior adds its interface requirements for a special behavior that translates the tree into actual widgets.

Other times, stronger coordination is needed. For instance, lenses need to compose effects where they overlap. The core system provides a mechanism for behaviors to request a unique (singleton) instance of a named behavior. In the case of lenses, this *manager* behavior computes lens intersections, and for each unique intersection recursively draws the document in that intersection or passes the event to the lenses active on that intersection. This manager behavior is also responsible for updating the stacking order of lenses; when a use clicks on a lens, it is move to the front and given first priority to affect painting and receive events. Another manager coordinates the user interface. Behaviors declare their user interface elements and semantic groupings, and the manager collects, sorts, and realizes the requests.

1.4 Contributions

The key contribution of this thesis is the definition and experimental validation of a digital document *architecture* that embraces the widest diversity and provides fertile ground for experimentation and subsequent distribution of innovations.

The architecture reifies and to an unprecedented extent opens the phases of the fundamental document lifecycle to extension. By highly factoring both document content and system functionality, as layers and behaviors—treating everything as an extension to a small core—the architecture boosts future innovations and work of interest to niche audiences to parity with the popular mainstream. Anyone can potentially benefit from inventions, and everyone is part of some niche audience.

In the generalized definition of document abstractions, the architecture is flexible enough to usefully support document applications as dissimilar as scanned page images and HTML, and provide an abstraction against which behaviors can be written and subsequently operate during runtime.

The hub document, a meta document format, is an essential element in the process of adding functionality to heterogeneous types, adding functionality to inexpressive types, composing content from disparate sources, and "editing" content on read-only sources. By describing the composition of components in the hub, individual components can be kept simple, and the final composition extensively customized.

Results of digital document experimentation from a potentially large number of third parties can be distributed widely, and integrated tightly and easily, in other Multivalent installations.

The three diverse applications of the architecture—enlivening scanned page images, extensible HTML, and distributed annotations in situ—validate the architecture, and promise an interesting space to explore other document types (PostScript, XML), media (video), data types (GIS, CAD), and interaction styles (collaboration), among others. The applications are also interesting in their own right, at places introducing novel capabilities.

Notemarks have been used for novel document visualizations, and they are readily available for new uses by others.

Multivalent is an architecture of possibility.

1.5 Overview of Dissertation

This dissertation follows a top down exposition from overall model through implementation to applications. Chapter 2 presents the fundamental document lifecycle that to some degree is shared by all digital document systems—restore, build, format, paint, events, clipboard, undo/redo, save—and its reification in the Multivalent model. Chapter 3 describes the other two parts of a digital document system, functionality and content, codified as behaviors and layers. Chapter 4 covers the supporting architectural elements to protocols, behaviors and layers: the document shell, the document tree, the graphics context, the hub document, and various other system support. Chapter 5 details implementation techniques necessary for good performance, scalability and robustness. Incremental algorithms during build, formatting, and painting are crucial for responsive interactive feedback with the user.

Chapter 6 describes three applications of the general architecture: enlivening scanned page images, extensible HTML, and distributed annotations in situ. The applications are diverse, yet not only is each possible within the architecture, but in fact the architecture enables novel, useful functionality, as demonstrated in the current chapter.

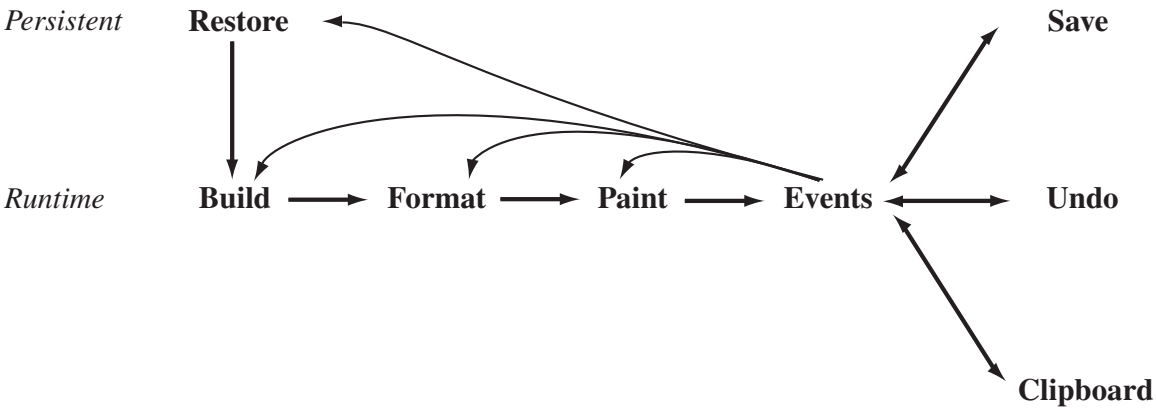
Chapter 7 compares other document models to Multivalent along several dimensions, such as whether extensions are finely integrated or merely juxtaposed with one another,

and whether extensions mutually compose together. Chapter 8 describes implementation experience and points to future directions.

2 Opening the Fundamental Document Lifecycle as Protocols

2.1 Fundamental Document Lifecycle

To some extent, every digital document system (editor, word processor, browser, viewer) implements what we term the Fundamental Document Lifecycle. As diagrammed below, the lifecycle concerns both persistent and runtime representations of the document in several phases. Generally and simplified for exposition without loss of essential qualities, a digital document system begins by creating a new document, or restoring one from persistent storage or perhaps from a computation that generates it on the fly. From the persistent or on-the-wire representation it builds an internal representation that is more amenable to programmatic manipulation, often a tree or graph data structure. Documents must be formatted or layed out, if they are not already layed out as stored. Formatting often includes but is not limited to placing images and other fixed-size media, and flowing and linebreaking text around them, all the while taking into consideration font sizes, styles, and proportional spacing. As the user scrolls through the document, it needs to be painted (also referred to as drawn or rendered) on the screen. Since painting needs to happen quickly in order to feel responsive to the user, painting usually involves as little computation as possible, using the geometries computed during formatting. At this point, the system waits in the event loop for events (mouse clicks, keypresses) from the user to direct what happens next.



The user sees the result of painting and, through the mouse, keyboard or other input device, dictates the next action to the system, an action which may involve any other phase. If the user requests a different document, perhaps via a hyperlink, the system loops back to Restore. If the user edits the document, this modifies the internal representation, which triggers some degree of reformatting and redisplay. Modifying formatting parameters found in a collection called a style sheet does not change the document tree but does require reformatting and redisplay. A number of events can require looping back and repainting: The user may scroll the document, bringing a new portion of the document into view. The user, through the window system of the computer's graphical user interface, may overlap another window on top of the document window, then bring the document window forward, thus necessitating repair of the formerly overlapped region. Action internal to the document, such a video clip proceeding to the next frame or a magnifying lens being dragged, can also require refreshing some part of the display.

Furthermore, with the system waiting in the event loop, the user can invoke various other standard services. The user can save the document to persistent storage, which writes out the document in a form that can be used to reconstruct at a later time the essential elements of current runtime representation. The user can undo or reverse some number of previous actions to recover from mistakes. Finally, the user can transfer document content to and from other applications via the system clipboard, onto which users cut or copy content from one source, and paste it to another. (In this conception of the lifecycle, printing is not a phase; it is considered part of painting, in which document content is drawn to paper rather than the screen.)

2.2 Extensibility and Composition through Open Protocols

In the usual digital document system, the implementation of these phases is hardcoded and, however powerful and configurable, is nonetheless fixed, and hence inherently limited. The Multivalent Document model takes each phase of the Fundamental Document Lifecycle and radically opens access to it at a fine-grained level, to program

code extensions called behaviors. Each phase is codified into a protocol that defines the flow of control behaviors should expect and the programmatic interface to which they must adhere.

At the highest level of abstraction, the Multivalent model directs the flow of control—at each phase and throughout the document—through self-proclaimed "interested behaviors" and gives them access to the same data structures as built-in implementations. Thus, behaviors are given equal power as built-in implementations, and can make arbitrary modification, large or small, to the "usual" (previously hardcoded) action of each phase. Furthermore, the Multivalent model defines flow of control through behavior extensions themselves, so that behaviors not only harmoniously modify a phase but so that the effects of numerous behaviors that all modify the same phase compose well one with another.

The Multivalent model defines a *framework*, as opposed to a code *library* or *toolkit*. Their difference lies in the locus of the flow of control. In a library, main control lies primarily outside, in user code, with the library serving as a useful set of functions. For instance, a program written in C may primarily solve a scientific problem, but periodically use the system input/output library. A framework, in contrast, directs the flow of control, periodically calling user code. Frameworks define architectures to which user code is fitted. However, frameworks are more difficult to establish:

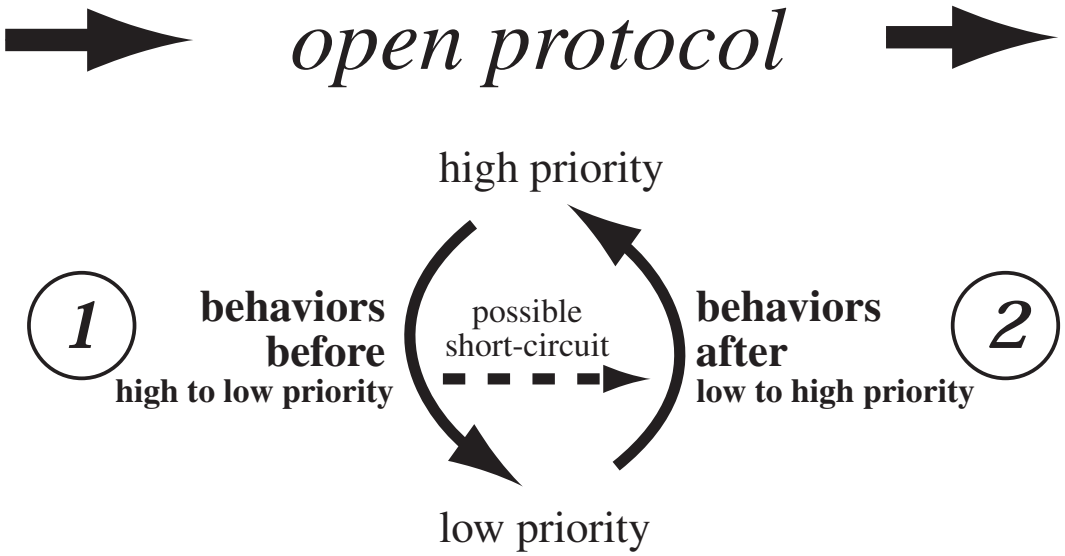
> If applications are hard to design, and toolkits are harder, then frameworks are hardest of all. A framework designer gambles that one architecture will work for all applications in the domain. Any substantive change to the framework's design would reduce its benefits considerably, since the framework's main contribution to an application is the architecture it defines. Therefore it's imperative to design the framework to be as flexible and extensible as possible. ... Frameworks often pose a steep learning curve to overcome before they're useful. [Gamma et alia 1995]

Bear in mind in the following arguments the Multivalent definition of extensibility: *power* to control and extend existing functionality, *flexibility* to accommodate unanticipated experimentation or innovation, and *composition* of extensions to produce a coherent whole at no disadvantage to monolithic systems conceived all of piece.

2.2.1 General Schematic

The diagram below illustrates the general way the Multivalent model transforms the phases of the Fundamental Document Lifecycle into open, programmatic protocols. After control flows in from another protocol and before it flow out to another, the open protocol precedes the usual action of its phase with an iteration through the "before" action of each "relevant behavior" and follows it with an iteration through the "after" action, where "usual action" (which may default to no action) and "relevant behaviors" are defined by each particular protocol. A before-after scheme can be found in the Common Lisp Object System (CLOS) [Kiczales *et alia* 1991] and to a more limited degree in Emacs [Lewis *et alia* 1995]. A before action can "short-circuit" control

directly to the corresponding behavior along the after chain, bypassing behaviors of lower priority as well as the usual action of the protocol. For those protocols that include a tree walk, a behavior can "register interest" on any number of nodes in order to take Before/After/Short-circuit actions on those nodes.

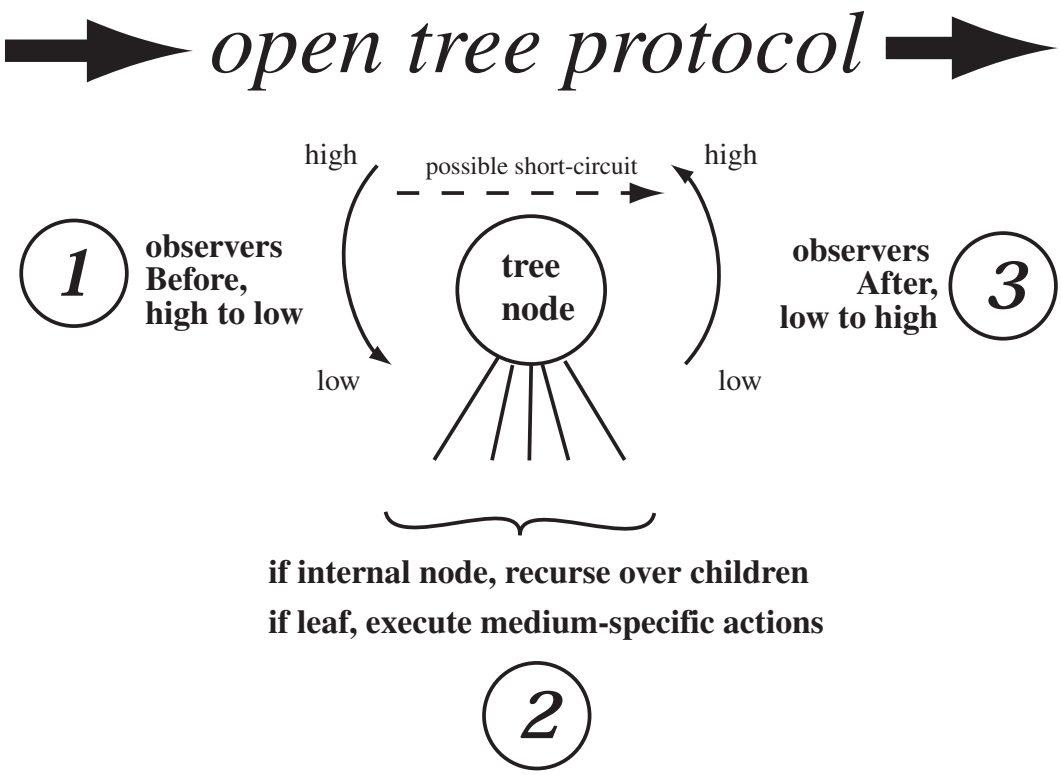


The global behavior priority dictates the order in which behaviors are executed. Behavior priority is initially set by the order in which behaviors were listed in the hub document (see Section 4.4), with subsequent behaviors receiving progressively lower priority, but this priority ordering may be changed. During the Before stage, which precedes the usual action of the protocol, control passes from highest priority monotonically down to lowest priority. Higher priority behaviors can establish conditions upon which lower priority behaviors can depend. At any point, a behavior may respond with a request to short-circuit the operation; if so, control bypasses any remaining behaviors in the sequence as well as the usual action of the protocol, and proceeds immediately to the same behavior in the After sequence. During the After stage, which follows the usual action of the protocol in the absence of short-circuiting, control passes from lowest priority to highest. Thus, After actions can assume that the Before and usual actions have been taken, and can modify or "massage" the results. Similarly, higher priority behaviors can massage the results of the after actions of lower priority behaviors. Informally, the highest priority "gets the first word and the last say".

2.2.2 General Schematic of Tree Protocols

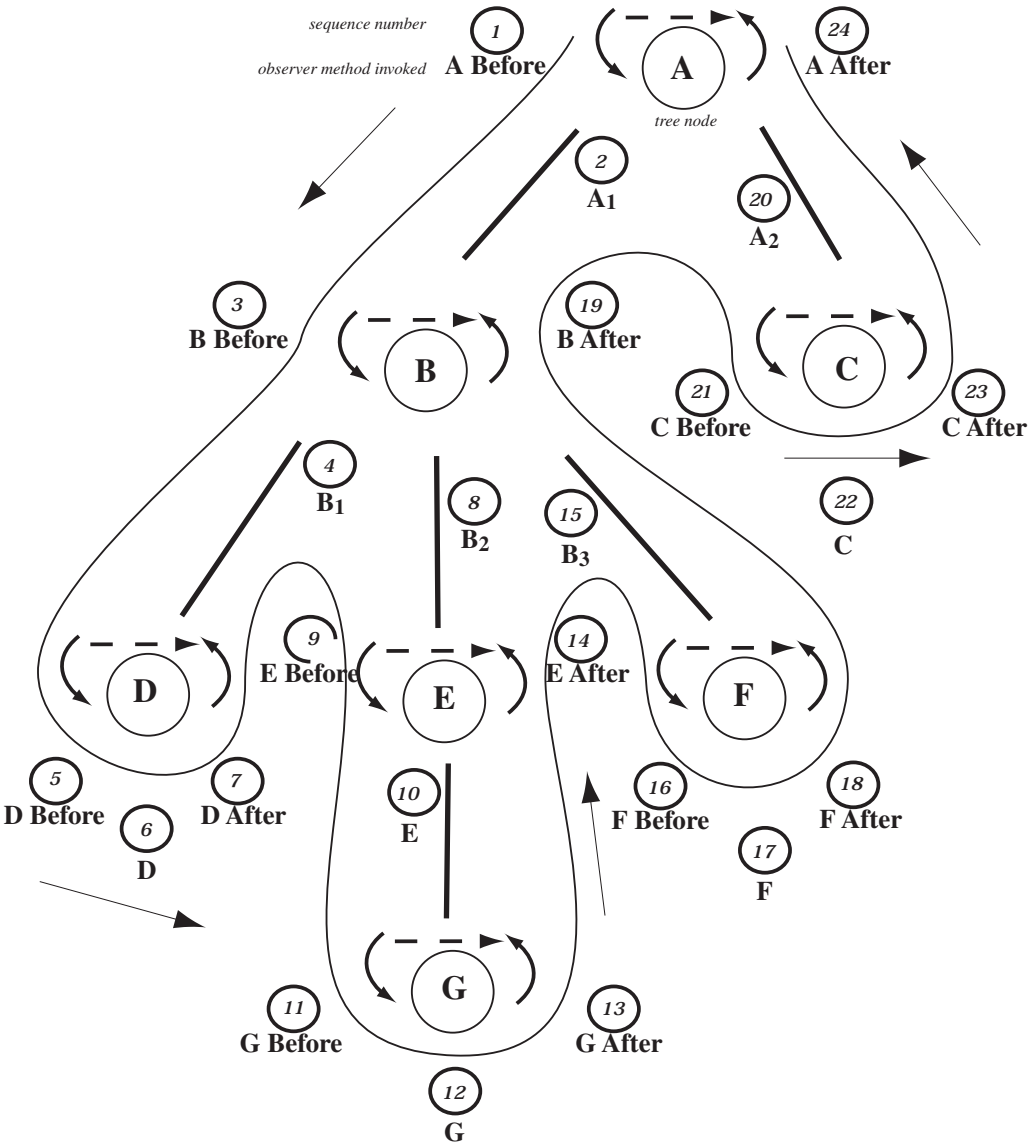
An important subclass of the generalized form of the schematic concerns protocols whose usual action involves a tree walk. These protocols are Format, Paint, Events, Clipboard. For them, interested behaviors can intervene in the protocol's action at any node or nodes

during the tree walk. As described in Section 4.2, “The Document Tree”, internal tree nodes usually take no action per se themselves, but pass on control to their children iteratively; and leaves implement medium-specific actions. A behavior “registers interest” in a node by adding itself to the list of observers kept by that node. The activity at each node is diagrammed below.



This per-node activity implies the following overall activity on the entire document tree.

A Tree Protocol Traversing the Tree

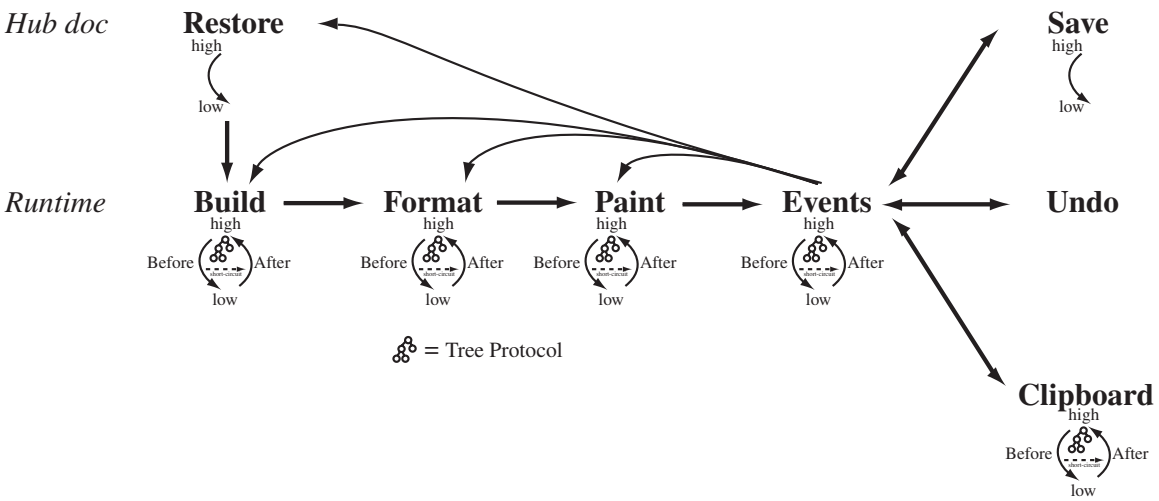


2.2.3 Summary

Because the Multivalent Document model opens each phase of the Fundamental Document Lifecycle in the schematic way described above, each protocol is extensible. Each extension may bypass or modify the usual action of the protocol at a fine-grained level. Moreover, behaviors can bypass or modify each other: they compose. Assuming behaviors take action appropriate to that protocol, the iteration according to the priority ordering of behaviors composes behaviors one with another. That is, behaviors compose together by virtue of adhering to the protocols: no specific mutual awareness is necessary.

This result is especially important because, once a system based on this model becomes widespread, authors will write behaviors without communicating with each other; yet these behaviors can be gracefully loaded into the same document as if the system were architected as a whole.

Each protocol specializes this schematic to some degree. The figure that follows summarizes the way the Multivalent Document model specializes each phase of the Fundamental Document Lifecycle. The form of each specialization and the reasons for it are considered at length in a following section devoted to that individual protocol.



2.3 Multivalent Protocols in Detail

The eight Multivalent protocols are considered individually, each illuminated as appropriate along six facets:

- Brief statement of purpose
- General characterization of appropriate behavior action
- Application to the running example
- Diagram of operation
- Algorithm in pseudocode
- Substantial examples

A brief statement of purpose summarizes in a few sentences the high level effect of the protocol.

The diagram and algorithm describe how the system framework invokes behaviors. Behaviors, in turn, must respond according to the general characterization of appropriate behavior action. It describes which actions belong in the Before subprotocol and which

in the After subprotocol, and what should be implemented by another entity, as by a tree node. Informally, successful implementation of the Multivalent model results from a cooperative handshake between the system framework and individual behaviors. This is a general characterization, describing the spirit of the type of appropriate actions and giving numerous examples; behaviors are open to various concrete document formats, media types, and even unforeseen areas, and thus by design it is impossible to definitively enumerate every action appropriate to every protocol.

An application to a running example ties together the effect of individual protocols. The subject of the running example is an ASCII document, elaborated in the Multivalent framework with highlights, hyperlinks, copy editor markup and floating notes. An HTML document or any other concrete document format would prove equally as instructive for the same protocols operate in the same way regardless of concrete document format; indeed, protocols should be written generally, against the abstract document tree, so that they operate well on any format. Among the simplest document formats, ASCII is an excellent pedagogic model, lacking the distracting idiosyncrasies of other, more complicated document formats, yet without oversimplifying any protocol.

A diagram of operation particularizes the general protocol schematic of the previous section to the specific protocol under consideration. It shows the behaviors involved, how the system framework invokes those behaviors, and how behaviors interact with the usual action of the protocol.

An algorithm describes with the detail and precision of program code the step by step operation of the protocol. The algorithm includes a statement of the "usual" work is for that protocol.

The general form of the Multivalent extensions to the traditional fundamental document lifecycle are simple and repeated with minor variation from protocol to protocol. Not immediately apparent are the powerful and versatile effects open protocols make possible. Thus, the detailed description of each protocol ends with one or more substantial examples of behaviors that exercise the particular protocol in non-obvious ways.

2.3.1 Restore Protocol

Purpose

The Restore protocol restores a document from a concrete document format and instantiates the building blocks of the runtime document. Often Restore reads a hub document specification (see Section 4.4) to determine the relevant behaviors and instantiates them with attributes from the hub by which behaviors can find their corresponding layers, if any. Usually layer data is cached in this phase, to be used in Build or another subsequent phase.

Description

Essentially the Restore protocol prepares behaviors and their data for use. As the system reads the hub document, it reads first any global document attributes, then instantiates (creates a runtime program object for) each behavior and invokes its Restore protocol method. Most protocols iterate over some subset of the behaviors comprising a document; the order of iteration is given, to a first approximation, by the order in which behaviors are listed in the hub document, with highest priority listed first. (Behaviors can be reordered, but this is uncommon.) A behavior's Restore method is invoked exactly once, and the behavior should at that time cache any data it needs. Besides loading of external data, any other time-consuming operations, such as initialization of data structures, should be done here.

Behaviors can be nested in the hub document; the system restores only the immediate children of the root of the hub, and these children can recursively instantiate their children. When the behavior is restored, it is given a handle to its children, if any, in the hub document. These may be inline layers, or other behaviors, which it can recursively, selectively instantiate. Supporting data layers may be restored from separate files or network connections or placed inline in the hub document. If placed inline, the data will be parsed into an SGML/XML parse tree, a structural tree similar to the document tree, with the nesting of the data exactly mirrored in the hierarchy of the tree.

Sets of heterogeneous annotations, as by a single author, can be grouped together and controlled as a single unit, which is in fact a layer. In executing protocols, the system iterates over a list of these layers, which in turn iterate over behaviors nested within; this process referred to by the phrase "iterate over document's behaviors".

Behaviors can be mutually coordinated in groups not specifically anticipated by the basic infrastructure. Instead of building support for lenses, for example, into the core infrastructure, lenses are coordinated by an ordinary behavior, known as a manager, which individual lenses notify of their existence during Restore. When a behavior needing coordination with a group of others is restored, it invokes a core infrastructure method with the name of the coordinating behavior. The system returns a handle to the single, shared coordinating behavior, instantiating it if necessary, to which the invoking behavior can then register its existence. In this way, behaviors can spontaneously generate new interest groups of mutually coordinated behaviors.

Running Example

The running example's hub document lists the following behaviors and associated parameters:

Behavior	**Attributes**
(top-level hub)	author, title, source
ASCII	URL of source ASCII text
Highlight (#1)	document location pair 1
Hyperlink (#1)	destination 1, document location pair 2

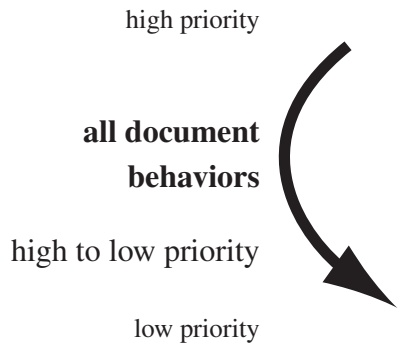
Note	x,y, width, height, unique root name, <textual content>
Hyperlink (#2)	destination 2, document location pair 3
Highlight (#2)	document location pair 4
CopyEdMan	
HyperMan	
HighMan	
NoteMan	
StandardFile	
StandardEdit	
DefaultEvents	

The Restore protocol first sets from the hub the global document attributes author, title, and source. The ASCII behavior is instantiated with the URL of the original ASCII document that is being embellished. In the Multivalent framework the ASCII text is called a layer, sometimes the "base" layer, where along with annotation layers, it is one of any number of other layers comprising the runtime document. The ASCII behavior reads in the content of the ASCII document and caches it for use in Build.

Highlight and Hyperlink annotations, not part of the base ASCII document, come with location attributes used to position them in the document. Since the geometric position of text may vary with such factors as screen size and available fonts, locations are given in document units like chapter and word (see Section 5.2, "Robust Locations"). As spans (see Section 3.1.3, "Span"), annotations have location pairs, a start and end position. Hyperlink holds additional behavior-specific information, namely the URL of the destination of the link.

The Note, with textual content, has a location as well, but given in physical coordinates, not structural. Other types of Notes can be placed relative to content and so have both semantic and geometric location specifiers. As a type of lens, Note registers its existence by invoking the lens manager, which is spontaneously loaded and not listed in the hub. Notes can themselves be annotated, and annotations need to find the correct document root when restored. Thus Notes place their unique root names in the global namespace paired with a handle to the instantiated root node. When saved, annotations save their documents root name, if it is not the default name (which is null).

Diagram



Restore

load behaviors and layers

Algorithm

```
set global document attributes taken from hub, if any
as each top-level behavior name and attributes encountered in hub document
   instantiate class of that name
   transfer hub attributes to instance's attributes
add to current runtime layer
invoke restore method
   (behavior initializes supporting data)
```

Example

The scanned page image application (see Section 6.1) uses a hub document to "bind" multiple pages into a unit, and exploits a hub's hierarchy to restore single pages at a time.

2.3.2 Build Protocol

Purpose

Build iterates over the behaviors instantiated by Restore to create the runtime document tree (see Section 4.2), the central data structure and means of behavior composition. Build is also responsible for the user interface tree, collecting together groups of related functions from disparate behaviors and organizing them into menus, on the toolbar, and in other composites. Finally, behaviors configure the system style sheet.

Description

The Build protocol iterates over the behaviors instantiated in Restore, passing them the root of the document tree from which behaviors can traverse to any part of the tree in order to add a subtree of content or mutate existing content. In the Before half of the protocol, behaviors augment the document tree with original content, with document structure reflected as tree nodes, content and metadata as leaves and node attributes. In

the After half of the protocol, behaviors can mutate the tree constructed in the Build Before stage and resolve saved location specifications to runtime tree nodes. They also report user interface requirements such as menu entries and tool bar real estate, which are sorted by the system into categories with requests from other behaviors. At the end of Build, the system creates the requested user interface elements.

The document tree constructed in Build mirrors the structure of the document. Behaviors are not allowed to modify the tree except to augment or correct structure. By guarding the tree from special-use hacks, all behaviors can reliably navigate the tree. From another perspective, the document tree is a canonical representation of the document, for some document formats resembling a parse tree, abstracted from the syntactic details of the concrete format, with structure explicitly exposed and easily navigable, and content and metadata easily available. (Aside from its primary use by subsequent protocols, the document tree makes a good representation with which to compare documents for differences.)

In the Build Before subprotocol, media adaptors (see Section 3.1.1) bridge their content from their data format into the document tree. Data formats remain encapsulated by medium, at the leaves of the document tree, leaving the internal tree uniform across media, This way, behaviors can be written against this abstract structure and yet manipulate any concrete medium. Media can keep sensitive content secure by contributing less to the tree, functioning as a black box to other behaviors, so long as they implement the lifecycle protocols, especially Format and Paint. However, media adaptor authors are encouraged to express the data format fully in the document tree, giving other behaviors maximum opportunity to manipulate its content in interesting ways. Theoretically, any structured medium, such as CAD diagrams, can benefit from exposing a structural representation in the Multivalent framework.

Behaviors that "hack" or mutate the work of media adaptors should not do so in the Build Before stage: the tree shape is a prime means for synchronizing layers, and if it were hacked by one behavior, subsequent ones would find greater challenge synchronizing. Instead, behaviors synchronized with the base representation of the document, such as table structure added to scanned page images, should resolve locations in the Before (not After) stage, and perform the actual hacking in the After stage.

Behaviors should also configure the system style sheet in Build Before. The high-to-low priority ordering of behaviors in Before lets the general settings by the media adaptor be overridden by personalized settings.

Behaviors can be notified of any protocol activity on any given subtree, and can modify or even supercede the protocol on that subtree. A behavior "registers interest" by adding itself as an observer on the subtree root (with the node's `addObserver` method). Subsequently, during protocols that involve a tree walk, the registered behavior and others so interested in that node have their corresponding Before methods invoked (Paint Before for the Paint protocol) before the protocol walks the node proper, followed by the corresponding After methods. The Before stage can establish conditions for the

subsequent tree walk, and the After stage can modify the results of the protocol accumulated on the subtree. The Before stage can also short-circuit directly to the After stage, bypassing the subtree entirely. This Before/After/short-circuit capability is available in most protocols; presently no implemented behaviors take advantage of it in Build.

Along with building the document tree, the other major function of Build is to construct the user interface. A single behavior can implement numerous functions and have numerous hooks in the user interface. All user interface needs from all behaviors are composed into semantic groups in a two-stage process. In stage one, during Build After, behaviors report the number of user interface elements required, their type (menu command, menu checkbox, menu radiobox, panel on toolbar), and their categories (debugging-related, annotation-related, and so on).

At the end of Build After, the system groups categories into menus and other user interface elements (categories can share menus, toolbars, and other elements), and instantiates them. The mapping from categories to menus can be given in the hub document or default to the mapping in the table below. Categories with a distinguished initial character (a slash, '/') result in cascading menus (submenus). Each menu automatically supplies a category of the same name. If a behavior requests a category name not in the mapping, a new menu is of the same name is created.

Menu	**Additional Categories**	**Comment**
File	BehaviorEdit, Print, Quit	
Edit	SelectAll, Search	
Style		Font family, style, point size
Go	GoIntra, GoInter	Intra- and interdocument navigation
Lens		Magnify, ShowOCR
Tool		Search, Spell check, …
Select	MathSelect, BiblioSelect, AuxSelect, MarkupSelect	
Anno	AnnoInk, /Note, /Layer	
CopyEd		
View	OCRView, ViewNB	
Meta	Affected, Debug	
Help		
Toolbar		not a menu

Behaviors are given links to their user interface elements and so can later modify the element's appearance in order to mirror document state (disable Save if the document is unchanged since the last save; show the current page number in a label).

Whenever the set of active behaviors changes, as when the user dynamically loads behaviors into the system, the document is rebuilt. Behaviors should maintain enough

information so that they can completely recreate their effect. For media adaptors, this can mean keeping a full copy of their data; alternatively, media adaptors can reload their data over the network, though this option is discouraged except for small data sets.

Running Example

For the ASCII document with behaviors instantiated during Restore, which are reiterated below from highest priority at top progressively to lowest, the system iterates over the list from highest to lowest priority invoking the Build Before action. In this case, only ASCII has any action, which is to add a structured representation of the text to the document tree. ASCII does not have internal structure beyond words and lines, but a more sophisticated implementation of the ASCII media adaptor may heuristically identify paragraphs.

In Build After, the highlight and hyperlink behaviors call upon the system's Location class to reassociate themselves in the ASCII document tree and attach themselves; their effect will be pronounced during later stages, in particular Format, Paint, and Events. (Notes do not attach themselves to the document tree itself.)

The second purpose of Build After is to construct the user interface. The table below lists each behavior's functions and each function's corresponding user interface categories.

Behavior	**List of Function / Category Pairs**
(top-level hub)	
ASCII	
Highlight (#1)	
Hyperlink (#1)	
Note	Toggle Note (shows note name) / Note
Hyperlink (#2)	
Highlight (#2)	
CopyEdMan	View Copy Ed as Notemarks / ViewNB, Short Comment / CopyEd, Insert Text / CopyEd, Insert Image / CopyEd, Replace With / CopyEd, Delete / CopyEd, Initial Cap / CopyEd, et cetera
HyperMan	Add Hyperlink / AnnoInk, Add Anchor / AnnoInk
HighMan	View Highlights as Notemarks / ViewNB, Add Highlight / AnnoInk
NoteMan	New Note / Note, Delete Current Note / Note
StandardFile	New / File, Open / File, Save / File, Save As / File , Close / File, Print / Print, Quit / Quit
StandardEdit	Undo / Edit, Redo / Edit, Cut / Edit, Copy / Edit, Paste / Edit, Clear / Edit, Select All / SelectAll
DefaultEvents	

Build Before collects functions by category, resulting in the organization below.

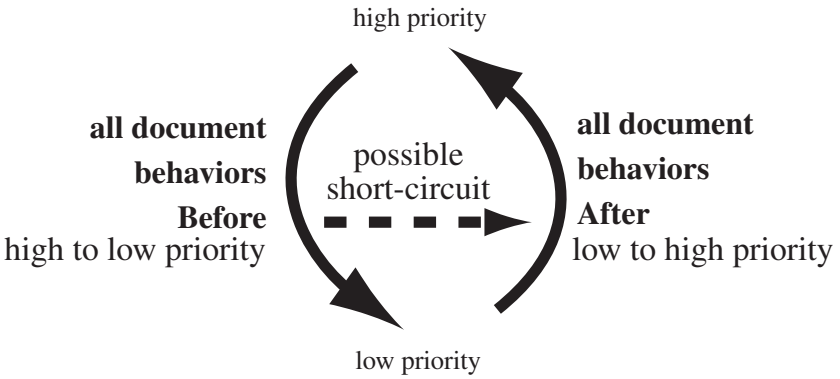
Category	Function
Note	Toggle Note (for particular note), New Note, Delete Current Note
File	New, Open, Save, Save As, Close
Print	Print
Quit	Quit
AnnoInk	Add Hyperlink, Add Anchor, Add Highlight
ViewNB	View Copy Ed as Notemarks, View Highlights as Notemarks
CopyEd	Short Comment, Insert Text, Insert Image, Replace With, Delete, Initial Cap
Edit	Undo, Redo, Cut, Copy, Paste, Clear
SelectAll	SelectAll

Categories here are sorted into menus according to the default table given in the Description section above since the hub document does not provide an organization, and the menus, toolbar, and other user interface widgets are finally constructed. If a menu group lacks entries, that menu is not constructed (there are no empty menus). Categories within the same menu are automatically partitioned by a separator bar. Radio items in the same category are mutually exclusive; that is, unanticipated behaviors can join and hence compose with existing radio groups. SelectAll isolates selection types from their the obvious grouping in File, as there are a variety of selection types (normal, tree structure, and others) that are mutually exclusive among themselves, but that should not be affected by other potential members of the File menu. It is common for radio groups to declare their own category.

Usually behaviors either contribute content or provide functionality through the user interface, but a behavior can do both. All behaviors in ASCII provide one or the other, but in the Xdoc behavior of scanned page images, the standard view of the document can be image or OCR (text), and Xdoc provides a menu item to toggle between views.

Notice that the standard File and Edit menus are not hardcoded into the system but rather are provided by an ordinary behavior. For the sake of simplicity, StandardFile and StandardEdit are shown as part of the running example's hub document, but in fact common behaviors can be factored out of individual hub documents into standard lists by genre, user, and system (see Section 4.4.1).

Diagram



Build Before
provide content

Build After
resolve locations, mutate content; build user interface

Algorithm

```
foreach behavior in high-to-low priority
   invoke buildBefore, passing root of document tree
       behaviors augment document tree
for each behavior in low-to-high priority
   invoke buildAfter, passing root of document tree
       behaviors mutate tree, resolve locations, report user interface

construct user interface based on categories collected in buildAfter
```

Example: Stripping Advertisements from HTML

The commercialization of the World Wide Web and the technical evolution of web browsers have brought with them a number of mechanisms for the invasion of privacy or attention: advertisements, blinking text, animated GIF images, and packets of data called "cookies" that can compromise privacy by, among other ways, tracking pages seen. Products have been developed to filter these annoyances from HTML, including interMute [Internet Mute 1998], which operates within the browser, and others that operate in the server or as proxy servers. The desired functionality is straightforward to implement, whether inside or outside of the Multivalent framework. Outside the framework, the most difficult aspect of writing such a filter is interfacing it with the browser; interMute installs itself as a proxy server that runs on the local machine. Installing the filter as a proxy server is straightforward, but this adds another step of mechanism and management to a feature that should feel like a built-in part of the browser.

In the Multivalent framework, a filter can be installed cleanly as an ordinary behavior. The functionality can be implemented simply, as a pass through the document tree as available in the Build After phase. Since all document content is available at that time

and behaviors can mutate the document tree, the filter simply scans the tree looking for undesirable elements (tree nodes and attributes) and edits them out or gives them benign effect (removing an advertisement might affect formatting, so it may be preferable to leave an empty box of the same size).

Filters during the Build After phase can perform any number of useful transformations. Time references could be translated to local time. Currency could be converted to local currency using current exchange rates fetched from a third party web site. Mouse button references could be swapped for lefthanders between "right" and "left", or "1" and "3". VCR-like buttons could be added to Java applets and animated GIF images to start, stop and pause them (rewinding general Java applets is technically infeasible). In the Multivalent framework, it is easy to install filters, to have multiple filters, and to configure them to taste.

Example: Augmenting Document Content

Like a digital Talmud (see Section 3.2), the content of Multivalent Documents is built up from multiple layers of content. For the running example, the ASCII document was given hyperlinks, highlights and Notes—capabilities straight ASCII is not capable of expressing—by augmenting the base document with additional layers.

Similarly, for scanned page images (shown in the Introduction and discussed at length in Section 6.1), mainstream page analysis yields partial information about the page: error-filled OCR, font attributes (bold, point size), possibly paragraphs and columns and tables. Additional analysis, by hand or with advanced algorithms that are perhaps more limited in the range of documents that can be analyzed, can extract additional structure such as the semantic components of bibliographic entries (author, title) or mathematical equations.

Rather than trying to retrofit the results of advanced analysis back into the original data format, which may or may not be capable of expressing it, this information is stored as separate layers and incorporated into the document tree during Build, making it available for manipulation by other behaviors.

Build lays the groundwork for table sorting and smart cut and paste for bibliographic entries and mathematics. A layer contains the physical location of rows and columns for table sorting, the semantic data of bibliographic entries, or the various translations for math. The layer also contains sufficient locations (see Section 5.2) to align this information with the document tree. Usually locations are taken with reference to the composite document tree, since that is the structure prevailing when annotations are made, and therefore locations are reattached during Build After, when the composite tree has been constructed. Analyses of the type discussed here, however, are often performed on the base tree, and so reattach locations during Build Before, before the base tree has been mutated. Since several layers of augmentation may be added, the tree is not mutated during Build Before, as this would disturb further reattachment (it is unnecessary to stress the robust reattachment mechanism here). Rather, all augmentations align

themselves during Before, and since reattachment points are robust to tree mutation, all mutations can robustly take place during After.

2.3.3 Format Protocol

Purpose

Format annotates the logical tree representation of the document with geometric positions. Paint uses these positions when making document content visible on the screen or printed page, and picking (which is not a protocol) uses them to map from screen coordinates to semantic document objects.

Description

Formatting executes along a depth-first tree walk from the root of the document tree. At every parent-child node pair in the tree, the child reports its width and height dimensions, and the parent positions it at an (x,y) location. The parent in turn computes its own dimensions as the union of bounding boxes of its children and passes this information to its own parent, awaiting positioning itself. For reasons of scalability (described in Section 5.1.2, "Incremental Formatting"), a node's coordinates are computed relative to its parent. Leaves are often specialized by media adaptors in order to report dimensions for new media, such as time, in an extensible way without hardcoding such knowledge into the core.

Internal leaves implement layout policies by implementing different algorithms to place children at particular (x,y) locations. The default layout stacks children vertically, like TeX's [Knuth 1984] vbox. The paragraph layout places children horizontally one after the other, linebreaking as necessary. Other layouts could be implemented to realize TeX linebreaking, lay text out top-to-bottom right-to-left or right-to-left top-to-bottom for various non-English languages, or produce jagged, jazzy Wired magazine-style layouts. Concrete document formats that have already formatted their contents, such as scanned pages and PostScript, are accommodated in the Multivalent framework through specialized tree nodes. To a first degree of approximation, these nodes simply accept the coordinates from the concrete format, compute bounding boxes for internal nodes, and transcribe from absolute to relative coordinates. Fixed format documents can be productively reformatted to some degree (see Section 6.1.3).

Thus layout styles are implemented by writing code, in contrast to declarative systems such as Proteus [Munson 1994] and the tree transformations of [Maverick 1998]. (Cascading Style Sheets [Bos *et alia* 1998] is not a complete layout system; it merely assigns values to HTML elements whose essential layout is hardcoded; it could not lay out a table under its own power.) As Maverick writes, it is a "standing engineering challenge to develop a presentation model that is *flexible*, … *powerful*, … and *efficient*". Implicit in this challenge is to maintain the simplicity that declarative systems can have over imperative programming. As declarative systems grow in flexibility and power,

they resemble more and more a general programming language. At the extreme, Document Style Semantics and Specification Language (DSSSL) [ISO/IEC 1996], with an elaborate layout model and powerful language, is effectively the programming language Scheme with a document manipulation library. Which is the same as provided in the Multivalent model: programming language and document model. At this point in the evolution of the Multivalent model, programmatic manipulation of the model is relatively close to the data structures. While layout per se is not a focus of the current work, it is sufficiently flexible and powerful as to not to have been an impediment to other goals. Still, declarative layout descriptions do have their advantages; such a system could be implemented on top of the mechanism provided here.

Graphics Properties Set by Style Sheet, Observer Behavior, Span

During the tree walk, the graphics context (see Section 4.3) holds the current settings of font, line justification, margins and other factors that affect the layout. Nodes are obligated to observe these settings in computing their layouts. Graphics context values are set by the prevailing active behaviors on that region of the document. The patterns of structure in the document, as encoded in the style sheet (see Section 4.5.3), contribute to the graphics context. Ad hoc behaviors such as lenses can override formatting in a subtree by registering interest at the root of the subtree. Spans can cut across strict hierarchical structure, spanning from a medium-specific point on one node to a point in the same or other node.

The tree walk begins with an empty active set and the style sheet. As the walk descends from parent to child, if the walk is entering a structural node, the style sheet is consulted for any relevant behavior, which if so is added to the active set; upon leaving the subtree rooted at that node, that behavior is removed from the active set. Whenever a behavior is added or removed from the active set, the graphics context recomputes its properties in priority order (see Section 4.3.2). Often some or all behaviors in a style sheet declare the same numerical priority. When a behavior is added to the active set, it is placed after all behaviors of equal priority, where it has final say on modifications, in effect giving it epsilon higher priority. Thus, in the top-down tree walk, style sheet settings for the more specific subtree override those set by its less specific parent.

Arbitrary behaviors outside of the style sheet can affect the formatting done by tree nodes. By registering interest on the relevant nodes as an observer, the behavior can participate in the active set. On structural nodes, behaviors function as ad hoc structural styles for the corresponding subtree. Upon entering a structural node, the Before methods for any ad hoc observers are called, any ad hoc or style sheet-associated behaviors are added to the active set, the children are layed out with this active set, ad hoc and style sheet behaviors are removed, and the After methods of ad hoc behaviors are called. For example, a Before method can “short-circuit” the process and proceed directly to the corresponding After method, which can be useful to efficiently process collapsed outline nodes, as they need not have their contents formatted.

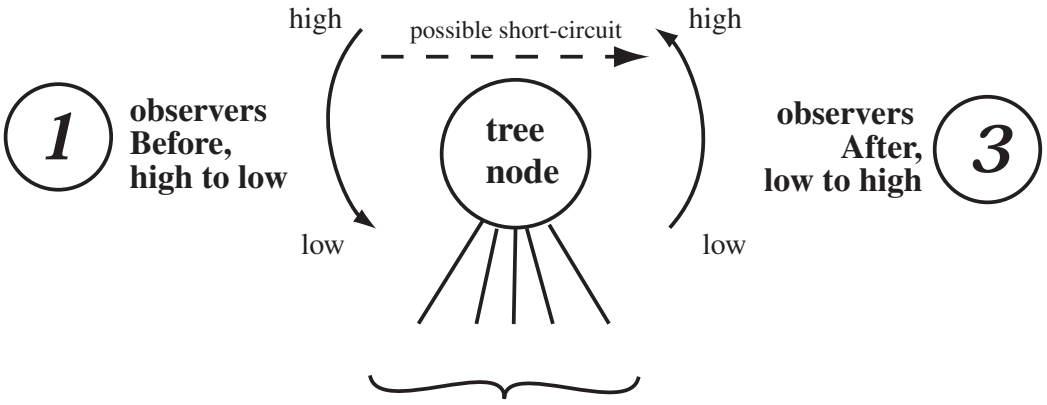
On a pair of leaf positions, a behavior can implement a span, where the first starts the span and the second ends it. Positions can be set internal to the node; the medium-specific node reports the number of internal segments it maps onto its medium, and positions are set on (left, internal segment) pairs. When the tree walk enters a leaf, it is segmented into homogeneous intervals between spans, and is traversed piecewise, with the active set updated at each transition.

Running Example

The most straightforward interpretation of an ASCII document is as a series of lines with no internal structure. It is different from scanned page images in that scans come, in effect, already formatted, with positions identified by the OCR. It is simpler than HTML in that HTML has tables, frames, lists, and other elements that take special formatting. HTML also has fonts of multiple sizes, but annotations can augment ASCII this way. In fact, ASCII could be augmented with images and even structure.

The Format protocol proceeds along a tree walk from the root. Upon reaching a leaf, which is a line of text (it could be a word), the leaf is required to report its dimensions to its parent. The media adaptor for ASCII consults the graphics context for the current font family, style, and point size, and computes the size of the text. If there are spans on the line, the media adaptor computes the dimensions piecewise, on each homogeneous piece between span transitions. The running example has two annotation types, highlights and hyperlinks, neither of which alters formatting. The Format protocol learns this by querying these annotations. The parent takes the width and height reported by the leaf and positions the line vertically, following any previous nodes under that parent.

Diagram



Algorithm

```
procedure format(Node n, GraphicsContext gc, StyleSheet ss, int maxwidth) {
   foreach observer on n, invoke formatBefore

   if (NOT shortcircuit) {
        /* descend into node */
        if (n matches pattern in stylesheet ss) update active set of ss

        /* process current node */
        if (n is internal node) /* actually implemented by subclassing */
               foreach child of n  /* depth-first traversal */
                      format(child)         /* determine width and height */
               foreach child of n
                      set child's (x,y) in relative coordinates to n,
                             mindful of maxwidth
        else /* it's a leaf */
               width ← 0, height ← 0
               while (span transitions on node)
                      process homogenous region up to transition
                             update running total width and height
                      foreach transition
                             add span starts to active set of ss
                             remove span ends from active set of ss
               set n's (width, height)

        /* ascend out of node */
        if (n matches pattern in stylesheet) restore active set
   }

   foreach observer on n, invoke formatAfter
}

format(
   root of document tree,
   empty graphics context,
   system style sheet
   width of screen
)
```

The determination of the leaf's width and height is determined uniquely per medium, where that medium respects the applicable property settings in the graphics context. Layout policies (horizontal placement, linebroken paragraph, table) are implemented in the (x,y) placement done in internal nodes.

2.3.4 Paint/Print Protocol

Purpose

Painting takes a formatted document and paints/draws/renders it on the screen.

Description

Painting is done in a medium-specific way, often through a media adaptor or a specialized leaf type. For instance, in a PostScript page, a word is painted by setting a font and painting the shapes corresponding to the characters, whereas in a scanned page image, a word might be painted by copying the corresponding part of the image of the full page.

In its traversal of the document tree and its handling of the graphics context, the Paint protocol is very similar to the Format protocol. To a large extent, inclusive of style sheets and spans, painting differs from formatting only in the action taken, drawing instead of positioning. In other ways, however, Paint emphasizes different aspects of the same mechanism than Format. The graphics context maintains a list of arbitrary name-value pairs that extend the standard set of attributes; behaviors that can respond to a signal look for it, and other behaviors can remain harmlessly ignorant. At least up to this point, signals have been used to modify appearance, drawing the content of a scanned page as OCR rather than the image, say.

Observer Before and After methods are more efficacious. Whereas Format's were used solely to trigger a short-circuit, Paint's are essential for such effects as underlining, side bars, and the range of copy editor symbols. The Before method, when used, often invokes a short-circuit after whatever drawing it does, as otherwise its effect could and probably would be painted over. The After method, which, according to the rule of thumb, assumes the usual painting has been done, embellishes it.

In the implementation, painting is done on a Java Graphics object (not to be confused with the graphics context). Ordinarily this Graphics object is an offscreen image, which is used to avoid flicker. However, painting can be done to any supplied Graphics object, including the one created by the Java PrintJob printing object. In fact, printing is implemented as an ordinary behavior, not as a core protocol, that manages Format and Paint.

Running Example

As Paint proceeds down the tree, internal nodes add relative coordinates to the running total to compute absolute coordinates, which can be trivially translated into the needed screen coordinates. Consider as an example the indented text below. Assume that one leaf has two spans, a hyperlink indicated by the underlined text, and a highlight which would be given a yellow background but here is indicated by the shadowed text. Painting proceeds piecewise. The first piece, "Painting takes a " is drawn with the current font. The width of this piece is added to the leaf's x-offset to give the position for the next piece.

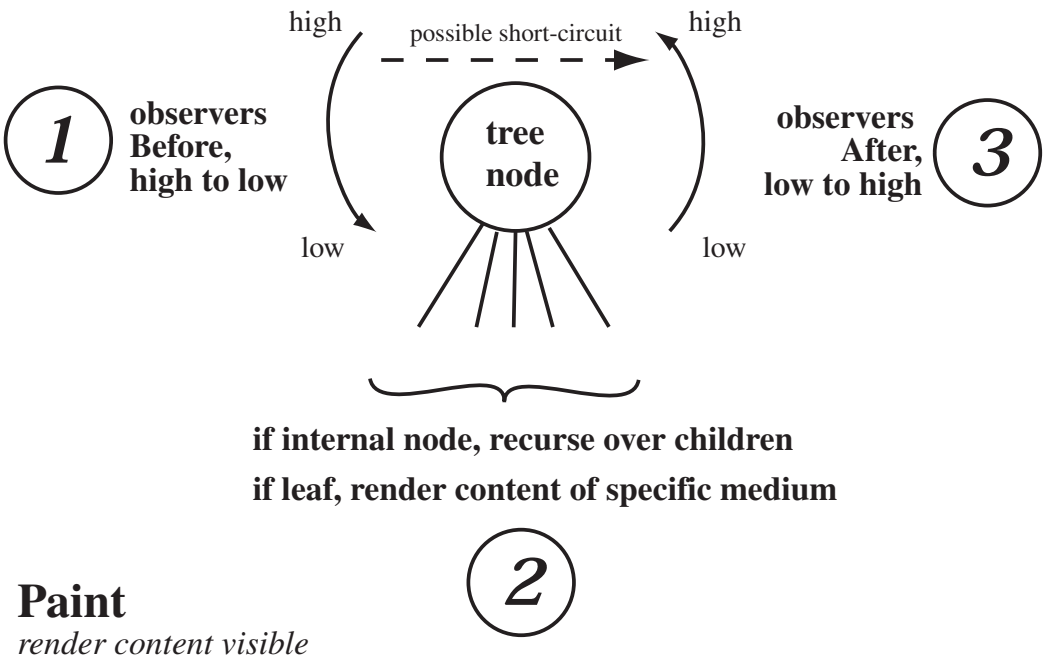
> Painting takes a formatted document and paints/draws/renders it on the screen.

Two spans start at the 'f' of "format"; the hyperlink sets the underline flag and sets the foreground to blue, and the highlight sets the background to yellow. If these spans

conflicted, if they both wanted to set the foreground color, for instance, both spans would still set that property, but the span with the higher self-declared priority would set it last and that would be the setting used by media adaptors.

At the second ‘t’ of “formatted” the hyperlink ends, at which point the hyperlink span is removed from the list of active behaviors in the graphics context and the values of the graphics context are recomputed based on the default values and the remaining active behaviors, which in this case is just the highlight. The segment of text “ted document” is drawn with a yellow background only. The final segment, “and paints/draw/renders it on the screen.” completes painting for this leaf.

Diagram



Algorithm

```
procedure paint(Node n, GraphicsContext gc, StyleSheet ss, Rectangle clip) {
   foreach observer on n, invoke paintBefore

   if (NOT shortcircuit) {
       /* descend into node */
       if (n matches pattern in stylesheet ss) update active set of ss

       /* process current node */
       if (n is internal node) /* actually implemented by subclassing */
              adjust clip coordinates to relative at this level of tree
              foreach child of n  /* depth-first traversal */
                     if (child bounding box intersects clip) paint(child)
              restore clip coordinates
       else /* it's a leaf */
              while (span transitions on node)
                     process homogenous region up to transition
                            /* paint medium-specific node content vis-à-vis gc */
                     foreach transition
                            add span starts to active set of ss
                            remove span ends from active set of ss

       /* ascend out of node */
       if (n matches pattern in stylesheet) update active set of ss
   }

   foreach observer on n, invoke paintAfter
}

paint(
   root of document tree,
   empty graphics context,
   system style sheet,
   bounding box of region to redraw (clipping region)
       in absolute document coordinates
)
```

2.3.5 Events Protocol

Purpose

Within the document display the user interacts with the system through and via events: key presses, mouse moves, mouse clicks, and so on. The Events protocol distributes events to interested behaviors. (Surrounding the document display, user interface widgets such as menus and tool palettes communicate directly with an associated behavior and do not participate in this protocol.)

Description

Events are delivered in a tree walk largely similar to Paint and Format and as described in Format, above. Before the tree walk, however, events are passed through lenses in the area. At this point in time, the magnify lens transforms the coordinates to correspond to the magnified space (it halves both x and y relative to the lens origin). Furthermore, if a

behavior, node, or span has set a grab (see Section 4.5.2) on events, it is given the event directly. Then spans in the area are offered the event. If none of these consumes the event, it is passed along the tree from root to leaf under the event.

If the event reaches a leaf and the leaf doesn't consume it, it is usually offered to the leaf's media adaptor.

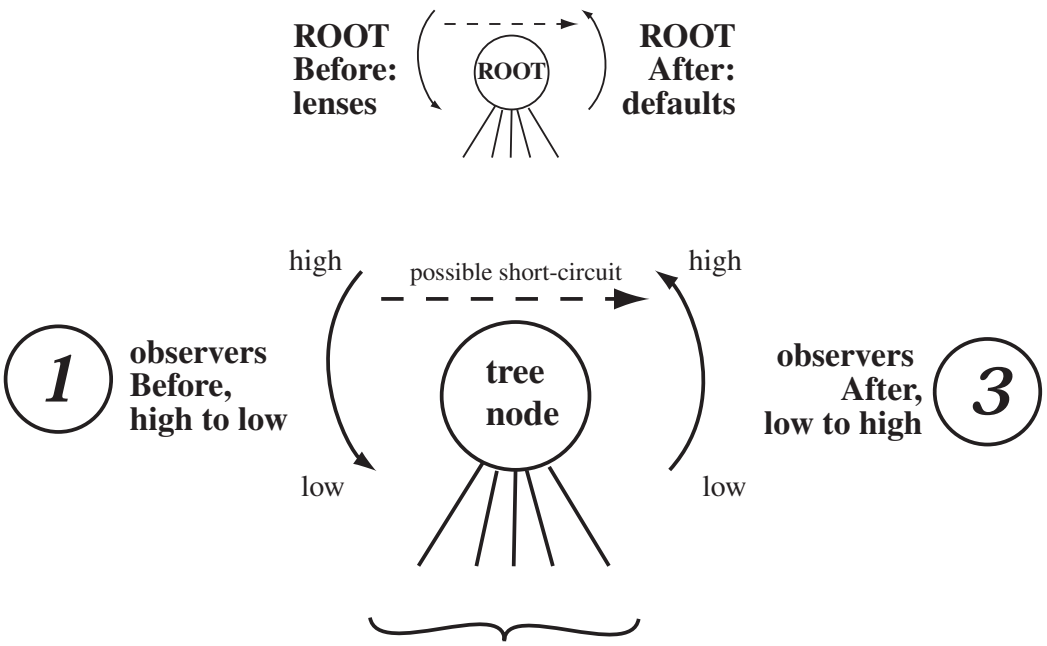
Standard key and mouse bindings are not hardcoded but rather are implemented via behaviors. These behaviors typically are observers of the root and consume events that have not been handled by any other entity; to receive all parts of multipart events, such as `MOUSE_UP` – `MOUSE_DOWN`, these behaviors will set a grab after receiving the first in the sequence. Since bindings are just behaviors, emulating the binding set of popular editors is a simple as swapping in different behaviors in the hub document. Like other behaviors, these behaviors can be cascaded so that a common behavior can handle events common to all editor emulations, events like scroll keys (page up/down, line up/down) and middle-button drag-scroll.

Running Example

The ASCII document has three regions that respond differently to input events: hyperlinks, Notes, and the rest of the document. These cases are considered below as examples in their own right.

Highlights participate in the event protocol, as do all spans when an event occurs along their extent, but highlights do not claim the event but instead pass it along the event resolution chain. Where hyperlink and highlight overlap, depending on their mutual priorities, either the hyperlink either seizes the mouse clicks it is interested in and prevents the highlight from seeing them, or the highlight is offered the event first but does not claim it, at which time the hyperlink does. Where the highlight extends past the hyperlink, the highlight again does not claim the event and it falls to the default event handler.

Diagram



Events

deliver event to highest-priority interested behavior

The tree walk for Events follows the same pattern as for other tree walk-based protocols. The traversal of the root node has been represented explicitly due to its important use in Events. As with all other protocols, the root is always traversed. Lenses, which can transform the coordinate space underneath them, operate in the Before subprotocol to warp the event coordinates (as of the mouse) to match that space so that back mappings from the screen to document structure work as expected. If after the entire tree of document content has been traversed and no behavior claimed the event, a behavior of default bindings can then claim the event without usurping more specialized behaviors. This is where scrolling with PageUp and PageDown keys, which is the desired action across many document genres, is implemented.

Algorithm

```
procedure treeEvent(Event e) {
   short-circuit ← false
   foreach observer on node    /* high-to-low priority */
       handleEventBefore, possibly setting short-circuit flag
       /* except for root, for which already executed (see below) */

   if (NOT short-circuit) {
       if (internal node)   /* actually implemented via subclassing */
              foreach child, treeEvent(e)
       else /* it's a leaf */
              compute active spans
              vis-à-vis old active span set,
                     send Leave event to spans leaving set
                     send Enter event to spans entering set
              old active span set ← current active spans
              foreach activespan
                     handleEvent, possibly setting short-circuit

              if (NOT short-circuit)
                     /* leaf medium-specific action, if any */
   }

   foreach observer on node /* low-to-high, starting at short-circuit, if any */
       handleEventAfter, possibly setting short-circuit flag
       /* except for root, for which already executed (see below) */

   return short-circuit
}



e ← given event from system
short-circuit ← false

foreach observer on document root
   handleEventBefore, possibly setting short-circuit flag
   /* special case (see Section 4.2.3, "Special Cases Managed by Root Node") */

if ((NOT short-circuit) AND grab set) {
   pass event to grabbing behavior
   set short-circuit flag
}

if (NOT short-circuit) treeEvent(e)

if (NOT short-circuit) {
   foreach observer on document root, handleEventAfter
}
```

Example: Default and Emacs Events

Events execute along a tree traversal but often no node or observing behavior is interested in the event. In this case, the event falls to the behavior DefaultEvents. This behavior registers interest on the root node so that it receives the event before the tree traversal (Event Before) and again, if the event is not consumed in the tree, after the tree traversal

(Event After). As its name implies, DefaultEvents takes action only if no other behavior has, and thus it operates entirely in Event After.

Among other actions, DefaultEvents implements the Selection. On a `MOUSE_DOWN` event, the Selection is cleared. If the cursor is over a leaf, the Selection is set to the empty span beginning and ending at the corresponding medium-specific segment within the leaf; the pivot point is set there too. During subsequent `MOUSE_DRAG`s, the Selection end is set to the span between the pivot point and the leaf segment under the cursor; if the cursor is not over a leaf, the Selection is unchanged. By virtue of being a span, when the Selection is screen is repainted continuously to match the span. Code common to all spans computes the difference between the old span position and new, and if there is overlap, only the difference is repainted; in the absence of overlap, the full old and new areas are repainted. The area to be repainted is computed as the lowest common ancestor node that includes in its subtree both start and end nodes. If the start and end points of the span cross a structural boundary, it is possible that the area repainted is larger than necessary, but in general this heuristic minimizes the number of relatively expensive tree traversals over the absolute number of pixels drawn.

DefaultEvents implements only those key and mouse bindings common to all editors. A user can easily choose Emacs key bindings or any other implemented editor by adding that behavior to the personal hub loaded for all document types. As explained in Section 4.4.1, “Cascading and Spontaneous Hubs”, behaviors on the personal hub have higher priority than those on the system hub, where DefaultEvents is found. Like DefaultEvents, EmacsEvents registers interest in the root. But since it is of higher priority that DefaultEvents, it receives the event first, thus the Emacs meaning of the event takes precedence over the default binding where there is conflict, and the event passes through to the default binding where there is no Emacs binding.

Example: Lenses

In addition to visually transforming an area in the Paint protocol, lenses receive the events that occur within them. Lenses register interest in the document root, receiving the event before the tree traversal (Event Before) and again after the tree traversal (Event After). Usually the lens lets the event pass unaffected.

The Magnify lens, however, transforms the coordinates to match its magnified display. For an event occurring at (x,y) in a lens with content area (exclusive of title bar) origin at (x_0,y_0), the event coordinates are transformed into $(x_0 + (x-x_0)/2,\ y_0 + (y-y_0)/2\ )$ before being passed on. With this transformation, event mappings from screen to document structure mappings operate correctly for selections and hyperlinks within the Magnify lens.

The Note lens holds its own document tree separate from the main document tree. It is opaque in its painting, and correspondingly it consumes all events it receives. Notes implements in-place editing of their content simply by claiming the cursor and selection from the base document (or other Note). Editing and span applications that take their

endpoints from the selection usually operate relative to whatever document tree owns the cursor and selection, thus once a Note does (as via a mouse click in the Note to set the cursor, with an additional drag to describe a selection), editing and span application operate within Notes without additional mechanism.

Example: Hyperlinks

A hyperlink is a span that is invoked by a mouse click—a `MOUSE_DOWN` followed by a `MOUSE_UP` without an intervening `MOUSE_DRAG`. A `MOUSE_DRAG` cancels the hyperlink and initiates a selection. Moreover, hyperlinks change the cursor shape when it is over the hyperlink and restore it when the cursor moves off.

The implementation of cursor changing is straightforward: the synthesized `ENTER` event from the system saves the current cursor shape, sets the hyperlink shape, and sets the system status message to the destination of the link; `LEAVE` restores the cursor shape and erases the message (sets it to empty). On a `MOUSE_DOWN` event, the hyperlink sets the system grab, sets its private active flag, and redraws the span in a different color (triggered by the flag) to indicate a pending hyperlink. A `MOUSE_UP` without intervening `MOUSE_DRAG` redraws the hyperlink in its original color, releases the grab, and calls the system to show the destination of the link.

An intervening `MOUSE_DRAG` initiates a selection. First it aborts the hyperlink by redrawing the link and releasing the grab. Then it synthesizes and sends to the document a `MOUSE_DOWN` event; as this event is seen by this very hyperlink, the fact that the hyperlink's active flag is still set indicates that it was self-generated and should be ignored. Finally, the hyperlink turns off its active flag and resends the `MOUSE_DRAG` event through the document; since the selection behavior has in the meantime set the grab, it receives the event directly.

Whereas HTML supports one type of link, the point-to-point link, whose Multivalent implementation is described above, the Multivalent framework provides fertile ground to implement link types investigated in hypertext systems and new link types, such as multi-tailed links which present the user with a choice of link destinations and links labeled to indicate their relationship to the current document (supports, contradicts) [Trigg 1983].

Example: Implicit Links

The Voyager Expanded Books [Voyager] line of digitized books aims to retain the affordances expected when reading a paper book and to exploit the computer to improve the reading process. One such improvement is the addition of implicit links such that clicking on any word presents a menu of commands vis-à-vis that word (such as finding the first occurrence of the word in the text). Similarly, the Oxford English Dictionary's interface [Tompa *et alia* 1993] passes the click to various independent applications for specialized handling.

Implicit links could be implemented in the Multivalent framework as a span covering the entire document or, alternatively, as a behavior registered on the root, where it would see every mouse click. Behaviors with functionality appropriate to a menu for implicit links could define a common user interface menu group. On an invoking click, the manager behavior coordinating implicit link-aware behaviors could first resolve the click to the corresponding word, search forward and backward to determine the word bounds, and pop-up the menu.

2.3.6 Clipboard Protocol

Purpose

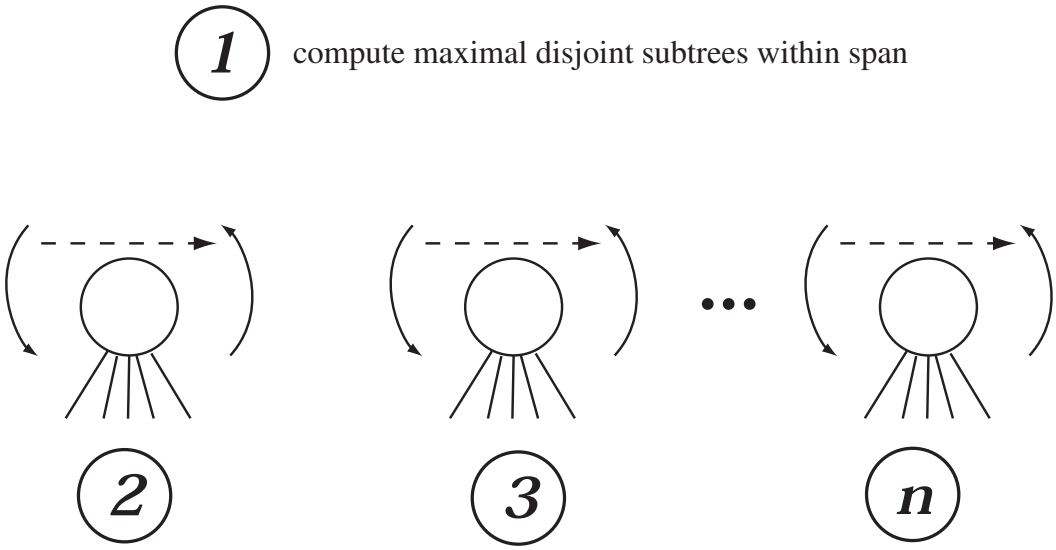
The Clipboard protocol takes the portion of the document described by the selection and places some representation of it on the system clipboard, from which it can be taken and pasted into another location in the same or other Multivalent document or any other application supporting the system clipboard.

Description

Clipboard constructs the selection's representation via one or more tree walks. As in Format and Paint protocols, the root of the document tree is a special case included in all tree walks (see Section 4.2.3, "Special Cases Managed by Root Node"); its Before method begins the walk, and its After method ends it. Unlike Format and Paint, which prune the tree talk against a geometric clipping region, Clipboard prunes against a distinguished span, called the selection. Clipboard converts the sequence of leaves in the span into a more structured representation, called a "chunky span", by replacing maximal sequences of leaves in the same subtree by the node at the root of that subtree. A chunky span in effect makes the span structured, so that structure-based behaviors can modify the default action along the span. In the general case, this yields a forest of internal nodes, representing subtrees, and leaves. Clipboard traverses each subtree in the forest from root to leaves. At each node, it calls Before methods of any observers, recurses to the children, then calls After methods.

Clipboard passes a shared string buffer along the tree walks that accumulates the representation of the selection. Ordinarily, internal nodes add nothing themselves but recurse to their children, and leaves as specialized by their medium append a string version of their content. Observing behaviors use Before/After and short-circuiting to achieve special effects.

Diagram



Clipboard

content transfer among applications

Algorithm

```
procedure treeClipboard(StringBuffer sb) {
   short-circuit ← false

   foreach observer, invoke clipboardBefore(sb), possibly setting short-circuit

   if (NOT short-circuit) {
       if (internal node)          /* actually, internal and leaf distinguished via sublclassing */
              foreach child, treeClipboard(sb)
       else /* it's a leaf */
              /* add to string buffer sb in medium-specific way */
   }

   if (NOT short-circuit)
       foreach observer, invoke clipboardBefore(sb}

   return short-circuit
}



sb ← empty string buffer to hold content of selection
short-circuit ← false
compute chunky span

if (root not in chunky span)       /* that is, if span not entire document */
   foreach observer of document root     /* low-to-high priority */
       invoke observer's clipboardBefore(sb), perhaps setting short-circuit

if (NOT short-circuit)
   foreach subtree root in chunky span
       treeClipboard(sb)

if ((NOT short-circuit) AND (root not in chunky span))
   foreach observer of document root     /* low-to-high priority */
       invoke observer's clipboardAfter(sb), perhaps setting short-circuit
```

Example: Provenance

Provenance augments the selected text with a description of its source, author, title, page number/URL and so on. Provenance operates by registering interest in the document root. By the time its Clipboard After method is invoked, the "usual" representation has been constructed. Provenance inspects the global document attributes for author and title, formats them, and inserts them at the beginning of the buffer. Thus, Provenance composes with other Clipboard observers as it takes what they construct and augments it.

Example: "Smart Cut and Paste"

When using the clipboard to move content internally or to an editor with a richer model than plain ASCII text, it is beneficial to communicate at that higher level and pass along text attribution and other media types. A behavior can entirely bypass the usual tree walk under system control and substitute its own representation. It does so by registering interest in the root and using its Clipboard Before to short-circuit the flow of control to its

Clipboard After. Within either the Before or After methods, this behavior can make its own tree walk and accumulate richer information.

A simple variation generates a structured, XML-compliant representation by writing out the label of internal nodes, recursing through the children, then writing out a corresponding close tag when the walk returns to that node. It calls the standard clipboard method at the leaves, and filters XML meta characters.

This scheme can be generalized to produce representations for any concrete document format. It is nontrivial, however, to take the Multivalent document as composed from various media and possibly multiple concrete document formats and generate a facsimile in a particular concrete format, especially considering that it is quite possible if not likely that the document format cannot represent various custom behaviors—copy editor markup, say—even if it can recognize it. Yet this capability of taking the document tree as edited and writing out an updated copy is useful in saving a new copy of the document in its original concrete format as well as for clipboard transfers, and so some minimal support is doubly desirable. In writing a media adaptor, behaviors can adopt a policy of translating what they recognize and ignoring what it does not. Presumably, it would recognize what it generates and probably a handful of useful extensions such as hyperlinks.

Example: Bibliographic Items and Mathematics

A general principle of the Multivalent approach is that the more highly structured the document is, the more capabilities it supports. Bibliographic entries and mathematics are two such instances. When copying via the clipboard from a bibliographic entry in the document and a database, the Multivalent system can automatically convert the data into the proper format, whether that be BibTeX, Refer, or other; and when copying to another paper that requires a particular style for its entries, IEEE house style, say, it can automatically convert to that style. The Introduction shows an example in which the bibliographic entries selected in a scanned page image have been automatically converted into the concrete formats for BibTeX and Refer.

The behavior implementing this functionality registers interest on the structural node covering each bibliographic entry, and uses its Before method to construct the desired concrete format and add it to the buffer, and then short-circuit out the usual traversal of that subtree. If the original document format does not structure bibliographic entry, as the XDOC format of scanned page images does not, the behavior can restructure the tree so that it does; in the prototype, bibliographic entries for scanned page images were described manually, though work in document analysis suggests that bibliographic entries can be determined and converted to semantically meaningful information automatically. Other document formats may encode the data directly or associate a unique identifier that can be used to look it up in a database.

However the bibliographic entry is constructed, the various concrete formats can be generated trivially and the set of concrete formats can easily be extended. Bibliographic

entry support has been evolved into a specialized class hierarchy so that to implement a given concrete format consists merely of formatting on the fly the information of the entry's author, title, pages, date, publisher, and so on, appropriately for that format; it is a stylized series of print statements.

Similarly, when copying mathematics from a paper into a math processor such as Mathematica or Maple, a programming language such as Lisp, or a math-savvy markup language such as TeX, the Multivalent system can automatically convert the data into the appropriate format. Mathematics is implemented the same way as bibliographic entries, except that, due to greater complexity of translating mathematics, the translations have been computed offline, rather than on the fly.

2.3.7 Undo/Redo Protocol

Purpose

Undo reverses the last action taken by a behavior. Repeated undos reverse more actions in strict reverse sequential order.

Description

To adhere to this protocol, a behavior must implement the corresponding inverse of any operation that mutates the document. In order to undo the operation successfully, it is usually necessary to store state concerning the pre-operation state of the document. It is the responsibility of the behavior to store this state. The system framework properly sequences undo/redo requests among behaviors and distributes them to the corresponding behaviors. In this way, unanticipated behaviors can successfully participate in the Undo/Redo protocol as no specific knowledge of their internal operation need be encoded into the core infrastructure.

When a behavior performs a mutating operation, it notifies the system of this fact so that it can record the behavior on its undo stack. For some behaviors, Undo is more natural if consecutive operations are chunked, and undone together; for instance, when typing in text, many text editors and word processors chunk together the characters inserted so an undo deletes them all as a unit, rather than, tediously, one by one. Chunking is possible only in the absence of any intervening mutation by another behavior. Thus, when a behavior notifies the system of an impending operation, it passes a flag indicating whether it would like to chunk if possible, and the system returns a flag indicating whether chunking is possible.

Some actions inherently cannot be undone. This is doubly serious as it erects a barrier beyond which undoing cannot progress—thus rendering all operations up to that point undoable—as the inherently undoable action renders unknowable the document state crucially relied upon by other behaviors in their undo-ing. Although behavior writers are encouraged to make operations undo-able if possible, the system supports this necessary

evil. In such a case, the behavior notifying the document of the impending operation sets a corresponding flag. Due to the seriousness of the action, the system verifies with the user in a dialog box whether to proceed. If so, it returns a go-ahead code to the behavior. Since this invalidates any undo state up to that point, the system frees the associated state, which may be considerable, in itself and all behaviors.

To execute an undo request, the system looks up the behavior at the current position in the undo stack and invokes its undo method. The system does not discard the information at that position and it would be needed by a redo at that point. Behaviors maintain their own stacks, with undo state information. The system guarantees that the document state exactly matches that just after the execution of the operation to be undone and strict sequential ordering of undo/redo request that matches the original execution order. Upon receiving the undo request, the behavior executes the corresponding inverse operation. (Thus behaviors that implement multiple operations record, as part of the document state, identification of the particular operation.) With the document undone by one step, the system advances its pointer in the undo stack.

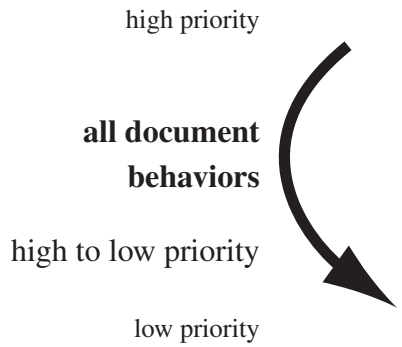
Redo proceeds identically to undo, in the opposite direction along the undo stack, except that the redo method is called. A behavior's redo implementation can usually use the same state as undo and call the original action, and thus are usually short wrapper methods that merely package the state as parameters to the original action.

2.3.8 Save Persistent State Protocol

Purpose

The Save protocol iterates over the behaviors corresponding to the runtime layers of the document, invoking their save method. Layers, in turn, iterate over the behaviors nested within them, invoking save. Ordinary behaviors save behavior-specific state in XML syntax. The result is a hub document that can be read back into the system later to reconstruct salient document state.

Diagram



Save

save hub document and behaviors-specific data

Algorithm

```
out ← open output stream
write Multivalent tag with global document attributes

foreach layer in document
   invoke layer's save(out)
   /* layer in turn usually iterates over behaviors in that layer invoking save */
```

2.3.9 Protocol Summary

Protocol	Type	Before action	After action
Restore	Layer	Read hub document Instantiate hub behaviors Hub behaviors load supporting layers	
Build	Layer	Construct main content of document tree Resolve locations keyed to unmodified base document	Mutate document tree Resolve locations Behaviors report user interface categories and types System builds user interface elements by category
Format	Tree*	Iterate over observer Before methods Internal node: iterate over children Leaf: report width and height requirements	Internal node: assign (x,y) relative coordinates to children Iterate over observer After methods
Paint/Print	Tree*	Iterate over observer Before methods Internal node: update graphics context with any structural information Leaf: update graphics context with any spans; draw medium-specific content	Iterate over observer After methods

Events	Tree*	[System: first any grab.] Iterate over observer Before methods Internal node: iterate over children Leaf: update synthesized span Leave and Enter events; medium-specific action, if any	Iterate over observer After methods
Clipboard	Modified Tree*	Iterate over observer Before methods Internal node: iterate over children Leaf: medium-specific addition to string buffer	Iterate over observer After methods
Undo/Redo	Direct	Undo/redo action in document tree	
Save	Layer	Iterate over layers in document invoking save method (layers' in turn usually iterate over their behaviors invoking save)	

* For tree traversal-based protocols, the root is a special case; its Before is guaranteed to be invoked before other action; its After afterward; and it can short-circuit between the two, bypassing the tree traversal entirely.

3 Highly Factoring Document Functionality and Content as Behaviors and Layers

Multivalent documents comprise functionality encapsulated into extensible behaviors and layers of content. Behaviors provide all the user-visible functionality in the system. Although every behavior has equal power to the others, classes of behaviors can be distinguished based on what aspects they exploit. The term layer refers to both a conceptual unit of document construction and a runtime data structure that treats groups of behaviors as a unit.

3.1 Behaviors

The core Multivalent framework performs no user-visible services on its own. Neither does it know about any specific medium. All user-level and medium-specific functionality is provided by extension code modules called behaviors. As the framework passes through the fundamental document lifecycle, it calls behaviors at the appropriate time for them to accomplish their functions.

Behaviors are implemented as Java [Gosling *et alia* 1996] classes, all derived from (inheriting from) a class itself named Behavior. There are class methods (functions) corresponding to the protocols, one for the Before half and one for the After, as appropriate. If a behavior wishes to participate in one or more protocols, it simply overrides the corresponding methods with the specialized implementations, accepting the default for the rest (the default usually is to do nothing). By adhering to the Multivalent

protocols, behaviors compose with the framework and with widely disparate behaviors, which are often written without specific awareness of one another. Following the principle that the amount of work to implement a new behavior is roughly proportional to the new specialized work provided by the behavior, the system implementation provides object-oriented classes from which new behaviors can inherit in order to gain the general behavior of the class.

The granularity of behaviors lies between the often short functions of scripting languages as found in, say, Visual Basic [Microsoft] and the nested editors of OpenDoc (see Section 7.1). Behaviors were conceived as the unit of extensibility, but the question arises of whether they are a usable unit of engineering for large software systems. Although the current implementation runs to 20,000 lines of code, not large in software engineering scale, the encapsulated quality of behaviors meshes well with that trend in computer languages and design methodologies, and in the current author's experience, behaviors have proven a comfortable unit of construction.

Although each behavior is potentially completely general in its operation, several classes have been found to be worth distinguishing. *Media adaptors* bridge between a concrete media format, such as PostScript or MPEG video, and the abstract document tree. One type of *structural* behavior builds up the semantic information of the document by elaborating the document tree, and another type is invoked more frequently to modify how another protocol operates in a document subtree. *Managers* coordinate select groups of behaviors more tightly than that provided by the general framework, coordination needed, for example, by time-based behaviors such as video and audio. An important subclass of managers serve as user interface intermediaries between the user and the creation of new behavior instances such as spans or lenses. *Spans*, such as the selection and hyperlinks, start at some point within a leaf and extend continuously in reading order through leaves to end at some point within a leaf, affecting appearance and receiving events along their extent. Finally, *Geometric* behaviors, or lenses, are subwindows that affect the appearance and events of a rectangular portion of the screen.

3.1.1 Media Adaptors

Media adaptors bridge between a concrete data format, such as PostScript or MPEG video, and the abstract document tree. They create structural tree nodes to match the structure of the medium and link them to the medium. Media adaptors concentrate their work in the Restore and Build protocols, and may contain state shared by their nodes Often a media adaptor will specialize (subclass) tree leaves, and less often internal tree nodes, to adapt them to the medium. Media adaptors are often closely related to their corresponding layer of information/data, providing no other function, as most non-media adaptor behaviors are general across media. Current media adaptors include scanned page images (with OCR supplied in XDOC format), ASCII, HTML as translated into XML, and UNIX manual pages.

Typically, media adaptors perform the bulk of their work in the Build protocol in augmenting the document tree being built with the media instance's structural subtree. They leave the tree-based protocols Format, Paint, and Events to the tree nodes they have created. Occasionally media adaptors specialize internal leaves, but almost always they specialize leaves that implement Format, Paint and Events for that medium. If the medium is entirely represented as a leaf, that is without accessible internal structure, it participates in the Format protocol by reporting its width and height dimensions. The leaf's parent will place the leaf at an (x,y) position. A leaf implements Paint by painting its content in the medium-specific way. For instance, drawing a word on an PostScript document entails setting the PostScript font and drawing the characters as PostScript shapes; drawing a word in a scanned page image entails drawing the corresponding portion of the entire page; and drawing a video entails decoding the frame corresponding to the current time and painting that image. Media without generally accessible internal structure can still allow the user to select subportions of the leaf by implementing an Events-related method that takes an (x,y) position within the leaf and returning to the caller an opaque cookie, a number that can be translated back to the substructural element when passed back to the medium-specific node, as for instance in a request to paint part of the leaf with the selection background color.

Some media adaptors can translate not only from media type to document tree, but from document tree to media type as well (in the Save protocol), although no currently implemented media adaptors do so. For instance, it would be useful to be able to edit an HTML document and save out a new HTML document. However, because the Multivalent document tree composes various media types and specialized behaviors, in general a particular concrete document format is not rich enough to represent the document tree. With the goal of producing the closest approximation of the document tree in the source document format, the knowledge of translating between document tree and data type could lie with the source format, which would encode a translation for each behavior, or in individual behaviors, which would encode a translation for each source format.

Rather than burdening the many behaviors with many translations, even for formats that cannot represent them, we charge the relatively small number of media adaptors with the translation burden. Thus, to save an edited version in a particular document format, that media adaptor's `save` method canvasses the tree and translates what it recognizes. As various document features such as hyperlinks become popular, it is increasingly likely that there will be a common implementation or at least a common base class that more adaptors will recognize and translate properly. Note that the saved document format need not be the same as the one from which the document tree was built; translations between formats can be performed by dynamically loading the target format and saving through it; in fact, the document tree is a high-information data structure, with both semantic, attributed structure and physical layout information, that makes for a good base for translation. This same mechanism can also communicate selections between aware applications at a higher level than the lowest common denominator strings.

Not only do media adaptors operate on media that have different representations, ASCII versus PostScript for instance, but they can add value to genres within a representation. The online documentation of the UNIX operating system, called manual pages, appear, to a first approximation, to be ASCII files when formatted. Yet within this formatting, additional structure can be derived and exploited. Although manual pages could be translated offline into a standard format, say XML, on-the-fly recognition of a format relieves the content source of creating and storing the alternative representation. This eases the examination of foreign repositories, which are unlikely to have the alternative representation.

Adaptors for media genres can exploit the structure in genre-specific ways. Manual pages can be tens of pages long, divided into sections and subsections. A manual page media adaptor could choose to build the page as a outline structured by sections and, nested within them, subsections. Manual pages have specific types of content unique to it, namely command line options and subcommands, that are likely to be reason for the user's browsing the page in the first place. Its adaptor can display these lines within otherwise collapsed outline sections, with a technique called Notemarks (see Section 6.3.3), to display a concise summary of the most important information in the page.

3.1.2 Structural

Structural behaviors operate on the semantic structure of the document tree. One type of structural behavior builds up the semantic information of the document by elaborating the document tree. This type concentrates its work in the Build protocol. For example, in scanned page images, standard character recognition software distinguishes between text and image areas, but does not analyze the page for higher level structure such as tables and does not have semantic knowledge of such areas as mathematics or bibliographic entries. Other document formats that focus on the content appearance rather than content semantics (which is then translated to appearance) also will not usually have semantic structures available for easy recognition and manipulation.

If a separate analyzer can derive semantic information in the existing content or additional content with semantic content is composed, a structural behavior can augment the document definition by adding the new information in the document tree. To improve existing content, it is usually necessary to change the tree shape to correspond to the structure. In the Build Before method, the behavior identifies the location of affected tree nodes (perhaps using a location object, see Section 5.2), and mutates the tree in Build After. This two-stage approach allows other behaviors that have saved locations keyed to the original tree to attach themselves correctly, as the location establish in Before is robust to any transformation short of deletion. If wholly new structural content is added (perhaps from another layer), the tree is mutated in the Build Before method as there was no content there before to which other content may have saved locations, and conversely, content that does have saved locations within the new semantic content needs that semantic content added as soon as possible in order to provide a place of attachment.

Another type of structural behavior is invoked more frequently to modify how another protocol operates in a document subtree. This type registers interest in the corresponding structural node, where every tree-based protocol calls the behavior's corresponding Before method, traverses the node and its children, then calls the After method, thus giving the behavior broad power to modify the protocol's result on that subtree. Smart select-and-paste, for example, seizes the Clipboard protocol during Before, computes the alternative representation, and bypasses the default text generation with a short-circuit.

Format and Paint protocols can be affected by a structural behavior, for special effects unsupported by the graphics context or that require computation; common structural formatting and painting effects are specified in the style sheet. Events could be intercepted at the root and transformed into one or more other events to provide a portion of a disabled accessibility package. For instance, input from a special input device could be translated into the keyboard and mouse events expected by the user interface.

Some document functionality uses both types of structural annotation. For document formats without explicit tables, table structure can be added by a structural behavior of the first type. Thereafter, a general table sorting behavior of the second type waits for a `MOUSE_DOWN` event, at which point it determines whether this corresponds to a column title and if so collects the content of the table, sorts it, mutates the tree to conform to the sorted order of table rows, then calls back to reformat and repaint the region. Likewise, a behavior of the first type could add abstract bibliographic information to a document. Thereafter, a general structural behavior of the second type computes the alternative clipboard representation for selected bibliographic entries of any document type, including those provided with the necessary semantic information (see Section 2.3.6).

3.1.3 Spans

Spans, such as the selection and hyperlinks, start at some point within a leaf and extend continuously in reading order through leaves to end at some point within a leaf, perhaps the same leaf. Spans can modify the current display parameters (the graphics context), draw arbitrarily, and receive user events. Spans save and restore themselves to and from hub documents. Usually, a manager behavior provides an interface for the user to dynamically add a span.

In addition to the protocols common to all behaviors, spans have two methods used during Format and Paint protocols. Spans report a self-declared priority distinct from the implied priority assigned by instantiation order. Ordinarily spans have priority higher than structural and style sheet behaviors but lower than lenses. In some cases it is necessary to override these default self-declared priorities, as when the selection declares highest priority so it will always be seen. Secondly, spans can be called several times along their extent in order to recompute the graphics context as other spans or other elements enter or leave the graphics context's active set. While Format and Paint methods could be used for this purpose, they are still useful in their regular roles in spans,

and so a new method (called `appearance`) is introduced whose sole purpose it to set properties of the graphics context.

While the document is being edited, spans on the document are guaranteed to maintain perfect alignment with the media they cover. Precisely what happens when text covered by one or spans is inserted and deleted is described in Section 4.5.5, "Editing". When a span is saved to persistent storage, it is augmented with multiple types of positional information so that if the base document is edited independently from the span, the span can usually realign itself correctly (see Section 5.2, "Robust Locations"). When restored from persistent storage, the endpoints of a span are usually resolved in the Build After subprotocol.

Spans of any length, from one character to the entire document, require storage for a pointer (actually, a Java handle) at the start leaf, another at the end leaf, and, as explained in Section 5.1.1, "Efficient Computation of Active Behaviors at Given Point in Tree", summary information in the tree of at most 2(h-2) storage units, less if the span covers less than the entire document, none if the span does not cross a subtree.

Painting Spans

When a span is added, deleted, moved (which is equivalent to a delete followed by an add), or its display properties change, incremental algorithms efficiently reformat and redisplay the smallest area required to reflect the change. Such a change marks as format-invalid all nodes from the span's start leaf to its end leaf and all their parents up to the root. As described in Section 5.1.2, "Incremental Formatting", the next time this area is to be painted—immediately if the span is visible on screen—the Paint protocol will detect the invalid states and reformat them. While reformatting, subtrees at any level that remain valid are reused, and a parent-relative coordinate system efficiently flows changed vertical dimensions through the remainder of the document.

When the document is formatted or painted and the tree talk reaches the (point within the) leaf that begins a span, that leaf adds the span to the graphics context's list of active behaviors contributing display properties; it is removed at the (point within the) leaf that ends the span. The graphics context resolves conflicts among behaviors wishing to control the same properties by priority. By default, spans have a (self-reported) priority higher than style sheet behaviors, since spans are overrides of the appearance given by style sheets, but lower than lenses, as lenses are more extremely ad hoc than spans. Style sheets, spans, and lenses can all modify any graphics context property, such as font, colors, and overstrike. Furthermore, during painting, when first added to the list of active behaviors, the span's Paint Before method is invoked, and when removed to the list, its Paint After, at which time arbitrary drawing can be done, with knowledge of its geometric position within the start and end leaves, respectively.

Span Events

Spans can also receive events. As described in Section 2.3.5, "Events Protocol", first lenses get a chance to handle events, then spans, then structural behaviors; this matches the priority rankings for setting graphical properties. Spans also receive two events synthesized by the infrastructure and delivered directly: Enter (span) and Leave (span).

A useful example of a span that responds to events is the hyperlink, where clicking the mouse on the hyperlink activates it. Its graphical properties (drawing the covered text in blue and drawing and underline, say) are implemented analogously to a highlight span. Upon seeing an Enter span event, the hyperlink changes the cursor to indicate to the user that the cursor is over a hyperlink; likewise, it changes the cursor back upon seeing a Leave span event. Upon seeing a `MOUSE_DOWN` event, the hyperlink changes its color and redraws itself, and sets a grab (see Section 4.5.2, "Grabs") in order to directly receive all future events, specifically the `MOUSE_UP`. Upon seeing the `MOUSE_UP`, the hyperlink releases the grab and jumps to the linked page.

Summary

In summary, spans exhibit the following stylized participation in protocols:

Protocol	**Action**
Restore	Ordinary participation
Build	Resolve start and end points during Build After (usually)
Format	Contribute to graphics context, with priority higher than structural but lower than lenses
Paint	Beyond graphics context, can paint has paintBefore and paintAfter with awareness of geometric position of endpoints
Events	Receive events, at priority lower than lenses and structural
Clipboard	Do not participate
Undo	Ordinary participation
Save	Ordinary participation

3.1.4 Geometric/Lens

Based on Xerox PARC's Magic Lenses [Bier *et alia* 1993 and Bier *et alia* 1994], Multivalent lenses affect the appearance of the geometric portion of the document under the lens. The user can resize, move, and change the stacking order of lenes. Where overlapped, lenses sum their effects, with conflicts resolved by giving the topmost lens highest priority. Section 1.3.1 demonstrates lenses.

Like spans, lenses have two methods beyond the fundamental ones implemented by all behaviors, a method that reports the lens's self-declared priority and another used to reset the graphics context when another behavior enters or leaves the active set.

Lenses are implemented as ordinary behaviors, without any new mechanism built into the core infrastructure beyond the Before/After/short-circuit already described.

Painting Lenses

Lenses are drawn using the "reparameterize and clip" scheme [Bier *et alia* 1993]. First the document is painted without regard to lenses. The lens coordination behavior (their manager) repaints those portions of the document covered by a lens. The coordination behavior registers on the root, as its Paint After method is invoked just before Paint would ordinarily complete, and repaints. It paints each lens once without regard for other lenses (except when a lens is entirely covered by others), and then computes lens intersections and paints each of those. For each lens of a collection of intersecting lenses, each lens involved is called for its Before method, then added to the active set while repainting the covered document content, then called for its After method. The After method usually paints the lens window control apparatus such as the title bar and resize bars.

Among lenses, priority order is determined by stacking order; the top lens received highest priority.

Pinned Lenses

Lenses can be pinned to the document and scroll along with it, or pinned to a fixed position the screen regardless of the scroll position of the document. The Paint protocol efficiently implements scrolls by less than a screenfull by copying the portion of the image that remains the same to its new position and drawing only the portion of the document newly brought into view. This would, however, incorrectly scroll lenses pinned to the screen. In this case, the lens coordination behavior exploits the flexibility of the protocol to correctly and efficiently repaint the entire screen. It maintains the scroll position of the last paint.

Since a lens is an observer on the root of the tree, it is called on any Paint. Its Paint Before method, which is called before drawing the content of the tree, checks to see if the scroll position has changed and it manages any pinned lenses. If not, control proceeds as usual. If both conditions are true, however, it reinvokes painting for the entire screen, and short-circuits the current paint.

Summary

In summary, lenses exhibit the following stylized participation in protocols:

Protocol	**Action**
Restore	Ordinary participation
Build	Not applicable (not attached to document tree)
Format	Contribute to graphics context, with highest priority, usually requiring (incremental) reformatting

Paint	Beyond graphics context, can paint has Paint Before and Paint After endpoints
Events	Receive events, at highest priority
Clipboard	Available, but no participation thus far
Undo	Not applicable
Save	Ordinary participation

3.1.5 Managers

Managers coordinate select groups of behaviors more closely than the general framework. For example, time-based behaviors such as video and audio need to synchronize on a common clock. Similarly, lenses need to coordinate among themselves in order to sum effects in lens intersections. The core framework has no awareness of time-based behaviors, lenses, or any other coordination group, but instead provides a means for the loosely coupled behaviors to synchronize on a single shared behavior. Behaviors request (via a method called `getSharedBehavior`) the shared behavior by name, and the method returns the shared copy, creating a new instance if none exists.

An important subclass of managers serves as intermediary between the user and the creation of new behavior instances such as spans or lenses. The copy editor manager and style manager are two such behaviors. The typical document contains many spans and a handful of lenses. For each span instance to have a menu user interface component would lead to much duplication in menus. Instead, a behavior manager exists solely to provide a user interface for creating new spans by the user. These managers know about spans to the extent that they can collect all the information required, such as span extent and possible text strings or other data, and create a new instance of the span. Subsequently, the manager has no knowledge of the span (except that it was created, so that the Undo protocol can do it). Spans save and restore themselves in the hub document, and provide an interface in the form of a pop-up menu, invokable along the extent of the span, to delete or otherwise edit themselves.

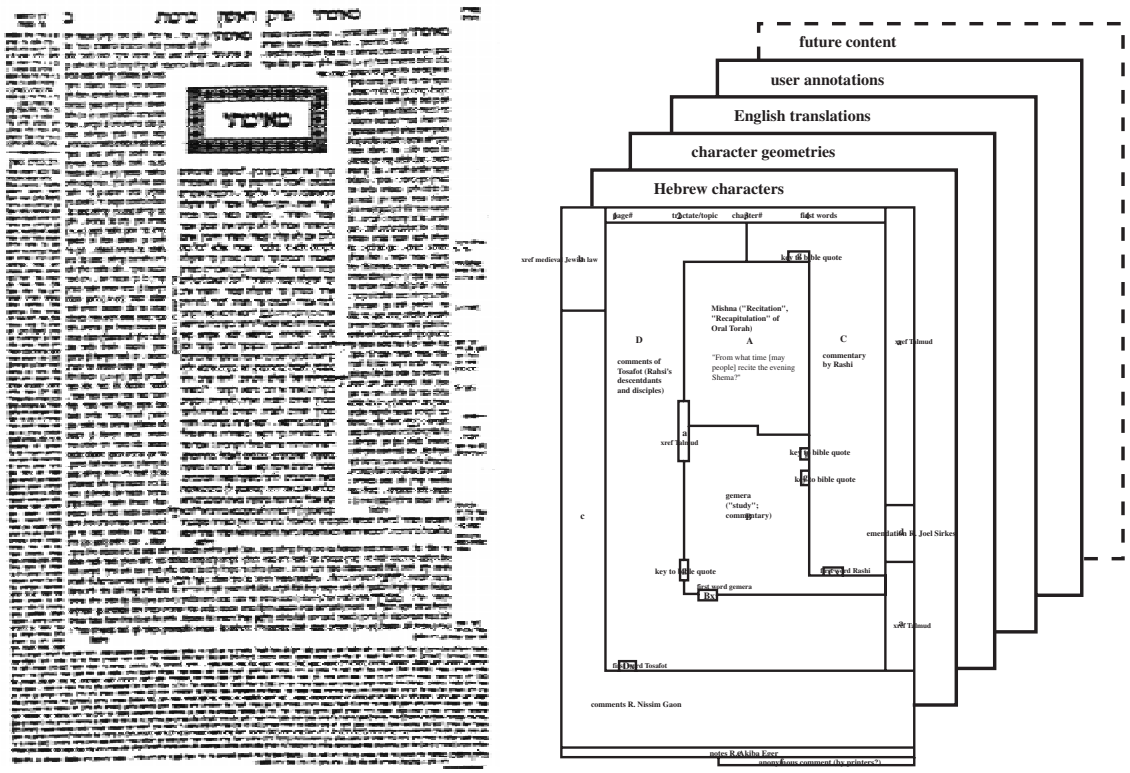
3.2 Layers of Information

The term *layer* is used in two different but related ways. Primarily layer refers to a conceptual way of building document content. Secondarily, layer refers to a runtime data structure that collects behaviors related in some way, for example as annotations made by a given commentator, in an often heterogeneous group. (Other systems apply this term to visual overlays; support for visual overlays is a forthcoming; see Section 8.2.)

3.2.1 Content Layer: Unit of Document Construction

Previously, most digital document systems were monolithic in that each new expressive capability of the document (as opposed to the supporting system), such as hyperlinks or video, was reflected back into the document format. Component-based systems such as OpenDoc improved the situation; however, individual OpenDoc editors were not extensible themselves—the features and limitations of one editor had to be exchanged wholesale for those of another (see Section 7.1). In contrast, the Multivalent model conceives documents as a set of semantically aligned layers. Individual layers, which can include existing concrete document formats, specialize in information content and compose together to result in a single conceptual document.

As an illustration of a layered document treatment, consider the Talmud, pictured below at left [Holtz 1984]. The Talmud is the written transcription of the early Jewish oral law. A digital Multivalent version of the Talmud could usefully model it as a number of heterogeneous layers. Since the Talmud's physical page layout has historical significance, the scholar may desire to work with the scanned image, but manipulate it in many of the same ways as the scanned page document demonstrated in the Introduction as well as additional ways that exploit the specific structure of the Talmud. To enable this, one layer could map the geometric regions of the page into semantic units, as the frontmost layer depicted below at right, units which include the *mishnah* (the single of the oral law on the page; labeled A), the gemara (the earliest Talmudic conversation; labeled B), the commentaries of Rashi (1040-1105) and tosafot (labeled C and D, respectively), cross-references to other passages in the Talmud, medieval codes of Jewish law (a and b, respectively), and comments by other rabbis (c, d, e), among others. Perhaps the page image has been shrunk to screen resolution (75-100 pixels per inch), but printing and a magnification lens could benefit from a second, higher resolution (say, 300 or 600 ppi) image. For manipulations such as selection and search that require geometrically positioned characters, supplied as two more letters. To make the text more accessible to non-Hebrew speakers, translations into other languages could be supplied, perhaps aligned with the Hebrew at sentence boundaries. In the Multivalent model, additional language translations could be added on demand; in fact, the user can add a translation without the knowledge or cooperation of the original digitizer of the text. The paper Talmud could hold only so much commentary as it was limited by the physical size of the page, but a digital version could add an unlimited number of commentaries, including personal notes. Finally, the Multivalent model is open ended, admitting future content to drive future behaviors.



Individual layers are kept simple, each contributing specific information content, without trying to be an all-encompassing format for all document features current and in the future. Additional layers can be added at any time, incrementally, and thus the document can evolve as document technology evolves. From another point of view, existing concrete document formats can be enlivened with new technology—features they cannot express directly, such as hyperlinks on ASCII or a scanned page image. Layers can be loaded from any source, from the local file system to over the network. That implies that some layers can be taken from read-only sources (a foreign server or a CD-ROM) but still augmented and saved to locations for which the user has read/write control. For the Talmud, perhaps the majority of layers is kept in Israel, but various vernacular language translations reside in the countries that speak that language. Layers can be authored by different parties; for the Talmud, one could even humbly add one's own annotation to the discussion.

Examples of implemented layers include the image, OCR text, and geometric character positions for scanned page images; HTML, ASCII, and manual page data formats, which are treated as base layers; and annotation groups (Notes, highlights, hyperlinks, copy editor marks).

Behaviors can exploit the information found in more than one layer. For instance, table sorting operates by mapping a mouse click on a column by the user looking at the image layer of a scanned page image to the column structure in another layer.

Layers may or may not have a visual component. Media adaptor behaviors often visualize their information, but in the case of table structure information, the layer is used to enrich the information structure of the document for use in manipulations by other behaviors.

3.2.2 Hierarchical Collection

The term layer also applies to a collection of behaviors that are in some ways manipulated as a unit, such as a set of annotations—call them commentaries—by a given author. Behaviors in the layer are loaded and saved together and can be disabled as a group (thus removing all the annotations by that author in one step). Somewhat confusingly, the runtime instantiation of this kind of layer is implemented by a behavior. Often the behaviors collected by such layers are heterogeneous, and in fact may include may include nested layers.

There is no fixed relationship among layers, files, and behaviors. A layer usually includes numerous behaviors. Data from one file may generate more than one layer; for scanned page images, the XDOC file produces both OCR and geometric positions conceptual layers. Data from multiple files may generate one layer, as annotations collected in one layer draw their content from the hub document and any number of additional separate files.

4 Supporting Architecture

Whereas the preceding two chapters, concerning protocols, behaviors and layers, describe the fundamental architectural structuring, numerous supporting pieces are needed in a complete working system. As mentioned previously, the core of the system is quite small and all user functionality is supplied by behaviors extensions. The document tree, the central data structure, is itself quite ordinary, but it is manipulated in a number of stylized ways. The graphics context is used to pass graphical properties along the tree during Format and Paint protocols. Hub documents record the components of stored documents and the configuration of the system itself.

4.1 Document Kernel

The document kernel is the fixed, initial bootstrap point that loads the rest of the system and subsequently manages the flow of control. As described in Section 2.2, “Open Protocols”, the kernel, or core, of the system is a framework, in which the general flow of control is fixed and from which behaviors are invoked according to protocols. Since a primary Multivalent goal is to support unanticipated extension, it is important that the immutable portion of the system dictate as little as possible—only that essential to compose the behaviors that perform the real work. The less that is immutable, the more that is customizable and improvable. Furthermore, since all behaviors are intimately defined by the structure of the kernel, it is important that the kernel not change much if at all, for a change to the kernel necessitates revision of potentially all behaviors. Although only widespread deployment will determine whether the kernel defines the appropriate abstractions, if the kernel small, there is less that can change.

The Multivalent kernel is quite small, about 1500 lines of code, compiling to about 40K of Java bytecodes. Primarily it manages flow of control through protocols. With regard to specific protocols, the kernel's operation can be summarized as follows:

> For the Restore protocol, the core reads the outermost level of a hub document, verifies that it is a hub document, and iterates over the behaviors listed in the next level of the hub, instantiating each with their attributes and content.
>
> For Save, the core writes the outermost level of the hub and iterates over all behaviors to write out their persistent data.
>
> During Build, the core iterates over all behaviors, which build up the document and user interface trees.
>
> For Undo, the core maintains the undo stack and calls the appropriate behavior with the stored state it needs to undo/redo.
>
> For the tree-based protocols, the core computes base parameter values and initiates a tree walk. For Format and Paint, initial parameters include a default graphics context and the clipping rectangle which needs to be filled.
>
> For Events, the core translates from window coordinates to absolute document coordinates; the core also manages event grabs (see Section) and synthesizes Enter and Leave events for spans, which are internal Multivalent constructions unknown to Java's event system.

The kernel also serves as a central communications point for all behaviors. Global state is available, including attributes and a pointer to the root of the document tree. Attribute values are not necessarily strings; for example, Notes are full document trees that can be annotated themselves, and Notes store their names and a pointer to their roots in the global namespace so that Note annotations can find them. Central services are found in the kernel as well, including opening a new document, layer management, and the painting to an offscreen buffer to prevent flickering. Behaviors request singleton manager instances from the kernel. When asked for a behavior by a given name, the kernel checks to see if it has cached an instantiation, and if so returns it; if not, it first instantiates and stores one.

Finally, the kernel mediates between Java and the computer operating system and the Multivalent model. The kernel itself follows Java's event model, whereas the Multivalent event model, while influenced by Java, is not the same (thus, the change in Java's event model from Java 1.0 to 1.1 necessitated a change in the core, but not in behaviors). Similarly, the core follows Java's clipboard model; for cut and copy, the core collects the information from the document tree and returns it to Java, and for paste, the core transfers information from Java system data structures to the document tree.

4.2 The Document Tree

The document tree is the runtime definition of the document content. It captures the structure and content of the document in a hierarchy of nodes. Internal nodes capture hierarchical structure, with typical names like "chapter" and "subsection". Leaves, specialized by medium, capture media content, preferably at a fine level of granularity: text (preferably as characters), pictures (preferably as geometric shapes), videos (preferably as frames or even something finer). All nodes are either internal nodes, which must have one or more children, or leaves. Currently tree nodes have both state and functionality; in the future tree nodes will hold state only, and associated behaviors will perform their functionality. As described in Section 2.2.2, "General Schematic of Tree Protocols", protocols involving a tree walk—Format, Paint, Events, Clipboard—pass the flow of control to the root of the document tree. Tree nodes implement a traversal that visits a node, visits its children, if any, then visits the node again; this is what is meant by "the usual action of the protocol", action that can be modified by behaviors associated with tree nodes.

Often internal nodes do no more than pass the flow of control on to its children or pass messages from its children one step further up the tree. The root of the tree is an internal node specialized to handle special cases. It is at the leaves of the tree that medium-specific action is taken, thereby isolating behaviors from medium-specific complexity. Thus, by writing behaviors to operate on an abstract document tree, they are guaranteed to operate on any concrete document format.

Sometimes media content does not directly map to a tree structure. A raw sound file may not have internal structure, and a table may want pointers along both rows and columns, which would produce a graph. A less preferred strategy places the content in a leaf; this would be appropriate for the raw sound file. For the table, it might not be important for one dimension to be explicitly represented; rows, say, can be directly represented in the tree with columns a subdivision thereof, and manipulating behaviors can use standard tree navigation operations to find the right child in the row for the desired column. The table implementation for HTML employs this method. Alternatively, specialized tree nodes could be defined for tables that add data fields for column traversal. What's important is that a well-define structure is established upon which all relevant behaviors can rely.

4.2.1 General Tree Node Properties

Tree nodes possess the following set of tree node properties.

Name	**Type**	**Description**
name	String	the structural name of a node; sometimes used by text leaves to store content
attr	Hashtable	Arbitrary number of name-value attribute pairs

observers	Vector of Behavior	list of behaviors that have registered interest in the node
parent	Node	tree parent of this node
children	Vector of Node	for internal nodes, list of child nodes
owner	Object	often the creator
bbox	Rectangle	bounding box of node and children
baseline	Integer	relative to bbox's y coordinate, the effective height of the node to use in laying out on a line
valid	Boolean	Is formatting valid?

If the source document format is SGML-like, the node name and set of attributes comes from the corresponding values; otherwise, structural facsimiles are constructed on a per-medium basis. Name and attributes can be read and written by all behaviors, and some behaviors take advantage of this to locate and validate structural entities such as tables and bibliographic entries or to change the appearance of the document by modifying attributes.

Behaviors can modify tree walk-based behaviors by *registering interest* in a node. In programmatic terms, this means that the behavior ask the relevant node to be added as an observer; nodes keep a list of observers and call them both before and after its children are traversed.

A node points to its owner, usually its creator behavior (its media adaptor), which may contain important shared state or provide specialized functionality, such as the medium-specific means of painting the content of the node.

The bounding box ("bbox") and baseline of a node are set during formatting. As described in Section 2.3.3, "Format Protocol", children propagate size information up the tree and parents set the (x,y) location of the node.

To enable partial evaluation of the tree, nodes are marked valid or invalid. As described in Section 5.1.2, "Incremental Formatting", nodes are validated before painting, invoking on-the-fly incremental formatting on invalid nodes.

4.2.2 Abstract Structural Tree Embodied in Internal Nodes

The document tree is a structural semantic hierarchy. A typical structural hierarchy for, say, a book divides into title page, parts, bibliography, and index, parts into chapters, chapters into sections, so on through subsections, subsubsections, and paragraphs, and within the bibliography, bibliographic entries with author, title, publisher, and publication date. This structure would be reflected in a tree by making chapters the immediate children of their corresponding parts in document reading order, sections he immediate children of chapters, and so on. Pioneered in Scribe [Reid 1980] and SGML [Goldfarb 1990], this is common practice for structured document systems.

To the degree possible for a particular concrete document format, the document tree captures the semantic organization of the document in the internal hierarchy of nodes, and the content of the document (text, illustrations, and others) in leaf nodes. If the source document format is SGML, or its equally expressive but more computationally tractable dialect XML, the shape of the document tree corresponds to the parse tree, with SGML generic identifiers becoming node names and SGML attributes becoming node attributes. The exception to the isomorphism between document tree and parse tree is in parse subtrees near the leaves that do not correspond to structure so much as make an ad hoc group in order to add a property, such as a hyperlink, to a range of content; these subtrees are instead flattened in the main tree and the property added as a span (see Section 3.1.3, "Span"). Document formats such as scanned page images and PostScript that have no semantic information, only geometric positioning information such as the coordinates at which to paint a particular text string in a particular font, are processed into a hierarchy that is semantic as much as possible. It is usually possible to obtain lines of text; sometimes it is possible to group lines into paragraphs and columns, and lines into illustrations; additional document analysis can deduce higher level structures.

The document tree is the central structure around which behaviors coordinate. It is the definition of the document content. When a behavior mutates the tree, the result becomes the new definition of the tree. For instance, table sorting is accomplished by extracting the content of the table, sorting it, mutating the tree to rearrange the rows, then reformatting and painting. Since the result of sorting produces a new definition of the document, other behaviors such as the selection and lenses perforce compose with the sorted table as nothing else exists that refers to the original tree.

Given the central importance of the document tree, great care is taken to maintain a straightforward semantic structure that behaviors can rely upon. No behavior is allowed to mutate the tree in a way that does not maintain or enhance semantic structure, whether to encode information, to increase performance, or for any other reason. For example, given a scanned page image with only geometric information, a behavior is allowed (encouraged) to apply the results of document analysis to group lines and words into, say, tables.

The system infrastructure does support one concession to efficiency. Some documents may generate structural representations with extremely "bushy" subtrees, that is, subtrees with many children. Very many children slows many algorithms, which examine all children of the nodes in a path from a given node to or from the root. Tk's text widget, which also relies on a balance between tree height and branching factor, "splits" a node at would-be overloaded points, inserting another tree level of children to which the former children at that level are apportioned. In order to more equally balance the children of a Multivalent internal node, the internal node may be split into a parent node of the same name and attributes and children with null names. The null name indicates a special case internal-yet-nonstructural node. The system provides convenience functions for navigating the tree and counting children that ignore the presence of internal nonstructural nodes. Unless a behavior specifically manipulates raw tree construction, it should use these convenience functions to observe a purely structural tree.

As presently implemented, the document tree contains geometric positioning and size information. Although this hinders multiple views and exotic formatting in which layout nesting does not mirror semantic nesting, it has a number of advantages, including simplicity.

Since internal nodes embody structural information and the style sheet (see Section 4.5.3, "Style Sheet") is structural, internal nodes manage communication with the style sheet. During tree walk-based protocols that use the graphics context, namely Format and Paint, internal nodes manage checking the style sheet for a matching structural pattern based on the node's name and its parents, adding any entry to the set of active behaviors for the context (see Section 4.3.2, "Setting Property Values"), and removing any entry as control passes out of the node.

4.2.3 Special Cases Managed by Root Node

The root of the tree is an internal node specialized to hold shared state and to handle special cases.

Since many nodes are created for a document, care should be taken to minimize their memory requirements. State shared among all nodes is kept in the root. Such state can include various attributes, as well as a pointer to the document data structure from which can be obtained document-global attributes and functionality, the current document URL (which is the hub URL if the document was not instantiated directly from a concrete document format), and document author and title attributes. The time to access shared attributes is proportional to the depth of the tree, as nodes climb successive parent pointers until reaching the root, and in practice speed is adequate as shared attributes are not heavily used (if access to any attribute becomes a bottleneck, it can be cached by the node).

The root node is not part of the document's structure per se and handles special cases. A following section describes one such special case, the unification of the several independent but coordinated visual layers.

If a behavior registers interest in the root, it will always be called for any action in the tree. Painting always proceeds from the root and so this is to be expected for paint, but this guarantee is useful for other behaviors, such as Clipboard, that can operate on subtrees or a forest of subtrees that do not include the root unless the entire document has been selected.

When the state of a node changes, perhaps due to editing or to the addition or removal of a span, that portion of the document needs to be repainted and perhaps reformatted as well. In order for it to operate correctly with lenses and other behaviors, painting proceeds top down, root to changed node. A node percolates its request to be repainted up to the root, which then translates the request into the proper clipping region for a call to the top down painting protocol. Similarly, when painting, the root singularly paints the

screen to the background color. This ensures that the entire background is painted as usually there is empty space not covered by document content, and it saves work as well as subtrees need draw the background only if it differs from the default (as with selected text). In another protocol specialized in the root, event coordinates are translated into absolute document coordinates, taking into consideration the document's vertical and horizontal scroll positions, before passing the event on to the rest of the document.

4.2.4 Medium-dependent Leaves Enable Medium-independent Behaviors

Leaves encapsulate medium-specific knowledge. As elaborated in the corresponding section on that protocol, leaves are black boxes that respond to the Format protocol by declaring their dimensions (to be placed by a parent), to Paint by drawing their contents vis-à-vis the prevailing graphics context, and to Clipboard by appending a representation into the clipboard. Furthermore, leaves report event hit detection within the leaf: the system identifies the leaf, and the leaf reports an integer cookie that it can be passed back at a later time to arrive at the same position within the leaf. For instance, within text, if the leaf contains a word, the subelement integer cookie might return the sequence number of the nearest character. To execute an operation in way tied to a specific medium, the leaf is specialized (subclassed), as described in Section 3.1.1, "Media Adaptors".

By encapsulating medium-specific knowledge in leaves, the exact same behavior extension code works on all concrete document formats, from rigid, preformatted scanned page images to hierarchical HTML and its window size-dependent formatting.

Just as internal nodes help keep the graphics context up to date by adding and removing style sheet-based behaviors, leaves do this for spans, which are anchored to leaves. Tree traversal across a leaf during Format and Paint proceeds by intervals divided at span transition points, span start or end points. At each transition point, the leaf removes from the list of active behaviors in the graphics context spans that end at that point and add those that start. Then the leaf formats or paints the subportion of the leaf from that point to the next transition point.

4.3 The Graphics Context

At any time when formatting or painting the document, the system needs to know the various graphical properties such as font, colors, and internal margins. These values can be set by structural properties, spans, lenses, and other behaviors. Conflicts between and among behaviors that attempt to set the same property are mediated by a system of priorities. Media adaptors are obligated to respect the settings in the graphics context as much as possible for that medium.

4.3.1 Standard Properties

The table below lists the standard set of graphics properties.

Property	**Description**
foreground, background	foreground and background colors
pagebackground	background color for entire page
justify	justification: left, right, center, fill
font	font object needed by Java...
family, size, style	... however font properties influenced separately
underline, underline2, overline, overstrike	colors to use for underline, a second underline, or overstrike, if any
elide	is text elided/invisible/hidden?
ydelta	gives delta position vis-a-vis baseline for subscripts and superscripts
xdelta	horizontal analog of ydelta (speculative)
xor	exclusive-OR mode
spaceabove, spacebelow	additional space
left/right/top/bottom margin	for structural nodes, …
x, baseline	reports current values to spans (see below)
signal	ad hoc name-value pairs for extending the property list (see below)

A few of the less common properties deserve further explanation. The font family, point size, and style (plain, bold, italic) can be influenced independently—taking the current font, say, Times-Roman, 24 point, bold, and adding italics to yield a bold-italic version of the same family and point size; the font field caches these properties in the Font object required by Java. Elide marks text to be ignored during formatting and painting; it is used for collapsed outline nodes, hidden speakers notes, and Notemarks (see Section 6.3.3). As formatting and painting progresses, two values, x and baseline, cache the current values of the current horizontal position in the line and its baseline, somewhat like the current point in PostScript; this is used by a number of behaviors, including the "short comment" annotation type, which draws its text starting at the span's beginning.

The signal field is used to extend these properties into new document functions. It holds name-value pairs that behaviors in those domains manipulate, query for, and respond to. For instance, the scanned page images media adaptor has the ability to paint the ASCII version of the content as well as the image representation of a word. It decides between these options by querying for the current value of the XDOC (named after the XDOC data format used) property. If it is IMAGE or nonexistent, the image representation of the word is drawn; otherwise the OCR/ASCII. The lens that shows the OCR of a region

simply modifies this property. In the future it may be necessary to develop conventions for naming signals to avoid collisions.

As well as the above properties that are available for public use by other behaviors, the context object maintains a list of behaviors active at the current point. Section 5.1.2, "Incremental Formatting for Editing and Fast Startup" describes how this list is efficiently initialized, and the section above on document tree leaves describes how it is maintained as new behaviors become active and inactive.

4.3.2 Setting Property Values

Graphical properties are set by structural behaviors, spans, lenses, and ad hoc overrides from any behavior. (A style sheet is a collection of structural behaviors active over the entire document. See Section 4.5.3.)

When one or more of these influences the same property, the property is set by the behavior with the highest priority; more accurately, the property is set by any behavior that wishes, but behaviors are invoked in priority order from lowest to highest, so the highest priority has final say. Behaviors that affect the graphics context have a self-declared priority distinct from their overall priority implied by the hub document (see Sections 3.1.2, 3.1.3, and 3.1.4). By default, lenses override spans override structural. This ordering yields matches that found in advanced word processors with styles, whereby the styles set the general appearance of the document, except where an ad hoc span, a boldface or italics word, say, overrides the style. Lenses, which are not commonly seen in document editors, are usually used to impose a special local, temporary, view and so override structure and spans both.

Nevertheless, any span, lens, or other behavior can declare any priority it wishes (with a method called `getPriority`). As an example of a behavior that declares a different priority, the span that shows the user's current selection declares itself to have maximum priority, so that it is always visible regardless of whatever other influences are in effect. Except in such special cases, behaviors can declare priority relative to others of its kind by reporting adding or subtracting system constants such as a `LITTLE` or a `LOT` to its base priority, such as `SPANPRIORITY`.

When two or more behaviors have equal priority, the tie is broken by giving implicit higher priority to the behavior that takes effect "later", where later for spans is that which starts at a later position in reading order, and later for lenses is the lens window on top. The HTML implementation, for which this model gives all tags have equal priority, also gives later settings higher priority. Tk's text widget [Welch and Uhler 1996], which only manages spans, avoids ties by imposing a total ordering and providing commands to raise and lower spans in the ordering.

4.4 Persistent Manifestation: Hub Document

The particular composition of layers and behaviors that comprise a given Multivalent document is captured persistently in a hub document. Written in the Extensible Markup Language (XML) [Bray *et alia* 1998], the hub document lists hierarchically the behaviors for that document, and provides the relevant attributes to behaviors to find their associated layer or layers or in some cases supplies layer content in the hub document itself. The hub document is not a new format so much as meta format that relates existing ones.

This separation of meta information in the hub from content and functionality has several advantages over more monolithic data encodings, especially ones tightly bound to a specific editor. For one, it is clearly extensible, as an author can merely list more behaviors and associated content, as opposed to a tightly circumscribed set of features. Hubs can compose unlike data formats that were meant and in fact may be hostile to composition (there is no place in scanned page image, ASCII, or a raw sound bitstream for copy editor markup). Since all layers need not be stored together, hubs enable spontaneous collaboration, without specific cooperation from the document source, which may be read only (a foreign server or a CD-ROM) or not supply annotation capability. Annotations and other augmentations to a document can be kept wherever write permission is available and the capability to view them is bundled with the hub. Finally, the hub encourages simplicity and orthogonality in individual layers, as each layer can focus on one job and do it well, and complexity is achieved by layer composition.

The order in which behaviors are listed in the hub document induces a priority ordering, where the first listed is given highest priority. Usually priority ordering among behaviors is inconsequential, except that the provider of the "base" document content should be prioritized higher than clients that manipulate that content. In fact, this implicit priority ordering applies strictly only to the top-level of the hierarchical document description. Within a subtree controlled by a behavior, that behavior can control whether and at what time to instantiate any behaviors in the subtree. This fact is exploited in the introduction of a paged document model to the system by an ordinary behavior with no special privileges. That is, the concept of a paged document is not fundamental to the framework. Instead, pages are subtrees controlled by a Page behavior that instantiates that page as appropriate.

4.4.1 Cascading and Spontaneous Hubs

The hub for a particular document does not list all the functionality realized in the instantiated document. Instead, they list only those ways they have been specialized compared to others of their type. Hubs cascade, or compose, starting with personal one for the current user, to a system one listing behaviors applicable to all documents, to a hub for the document's genre, to a fourth and final one for the document itself. Although

configuration may vary, typically the personal hub is kept as a local file, the system and genre hubs are kept on the server alongside the behavior code, and a particular document's hub is kept locally if edits and annotations are personal, on a shared server if the document is shared. The componentized organization of hubs yields the same maintainability and consistency benefits as style sheets for structured document. It also keeps individual document hubs small, listing only those ways the individual differs from its genre. In fact, if there is no difference, no hub is required, as one can be spontaneously generated (its type is heuristically deduced from MIME type or filename suffix) with no loss of functionality.

As an illustration of cascading hubs, consider the behaviors marshaled by the hubs below.

System	**Xdoc Genre**	**HTML Genre**	**Personal**	**Individual document's hub**
StandardFile StandardEdit Search Define Magnify DefaultEvents	ShowOCR ShowImage	HTMLFilter HTMLTableSort	EmacsEvents CopyEd HyperlinkMan HighlightMan NoteMan BiblioBibTeX SpeedRead Bed Debug	Biblio Hyperlink Highlight x 5 Note x 2

In the personal hub, in effect across all concrete document formats, the user has chosen Emacs editor keybindings. This user would like to have available at all times annotation capabilities, including copyeditor markup (CopyEd), hyperlinking (HyperlinkMan), highlighting (HighlightMan), and notes (NoteMan). Whenever bibliographic structural information is available, the user wants such selected text to be transformed into BibTeX format before being placed on the clipboard. The user has found helpful the particular speed reading scheme implemented by SpeedRead. In order to dynamically load new behaviors into the system, perhaps from third parties, or unload unused behaviors, the user lists the behavior editor (Bed). Finally, since the user is currently developing a new behavior, he has requested the various low-level system introspection utilities provided by Debug.

The system hub, in effect for all users across all documents, sets the standard File menu, with Open, Save, Save As, Close, and Quit commands; and the standard Edit menu, which Cut, Copy, Paste, Delete, and Undo commands, among others. Also ubiquitous are searching within the document, requesting word definitions, and enlarging a portion of the screen with the Magnify lens. Finally the system provides a set of default events that handle tasks common to all keybinding maps, such as selection when on mouse click and drag and page scrolling with PageUp and PageDown keys. (In the future, to

maximize customizability, perhaps the system hub will not be part of instantiation for all documents, but instead initialize the user's pan-document hub.)

The genre hub, in this example for the Xdoc genre used for scanned page images, supplies two lenses, ShowOCR for showing for its corresponding text under the lens instead of the image, and ShowImage, which reverses the operation. In contrast, these lenses would be worthless for HTML. Likewise, the HTMLFilter behavior, which removes advertisements massages the document in ways specific to HTML, is useful for all HTML documents, worthless for Xdoc.

Finally, a per-document hub describes the uniqueness of individual documents. In our running scanned page image example, the document has bibliographic structure (Biblio) derived, which can be utilized by others. Annotations are frequent source of a per-document hub.

If a (concrete) document were instantiated directly, bypassing the hub, the system would determine the corresponding genre, and cascade in the corresponding genre hub, if any. The effect is identical to instantiating a per-document hub that is empty except for the genre and a pointer to the corresponding concrete document. In this example, the set of behaviors loaded when instantiating the concrete document directly is identical to that retrieved by instantiating the individual document hub, except for the bibliographic manipulation and annotations. The user can then customize the document, perhaps with annotations, and save it as a hub to capture these differences from its genre.

4.4.2 Sample Hub Document

The following is an actual, complete hub document, slightly reformatted for readability. It was generated on the fly by a script that took title and author information from a bibliography data file and wrapped the nine individual pages comprising the complete document. Abstractly, hubs simply list relevant behaviors; this sample shows the syntax and auxiliary information contained in a hub.

```
<!--
   Document saved at Mon Feb 02 16:26:26 PST 1998
-->

<MULTIVALENT
   TITLE="A Model of Sales"  AUTHOR="Hal Varian"  SOURCE="X"
   GENRE="Xdoc"  PAGE="1"  PAGECNT="9"
>

<Multivalent.std.adaptor.Xdoc  BEHAVIOR="Multivalent.std.adaptor.Xdoc"
   XDOCURL="http://elib.cs.berkeley.edu/docs/data/0600/620/OCR-XDOC/%08s.xdc"
   IMAGEURL="http://elib.cs.berkeley.edu/docs/data/0600/620/GIF-
INLINE/%08s.gif">
   SCALE="3.3"
</Multivalent.std.adaptor.Xdoc>


<Pages>
<Page  BEHAVIOR="Multivalent.Layer"  NAME="Personal"  URL="inline"  NUMBER=1>

<Span  BEHAVIOR="Multivalent.std.span.HyperlinkSpan"  CREATEDAT="Mon Feb 02
16:24:17 PST 1998"
     HREF="http://www.sims/">
  <Start  BEHAVIOR="Multivalent.Location"
     TREE="0  2/Salop 1/PARA 4/REGION 0/OCR"  CONTEXT="Salop the and"></Start>
  <End  BEHAVIOR="Multivalent.Location"
     TREE="8  4/Stiglitz 1/PARA 4/REGION 0/OCR"  CONTEXT="Stiglitz and
model"></End>
</Span>

<Span  BEHAVIOR="Multivalent.std.span.AwkSpan"  CREATEDAT="Mon Feb 02 16:24:54
PST 1998"
     NB="COPYEDNB"  COMMENT="What%27s+this%3f">
  <Start  BEHAVIOR="Multivalent.Location"
     TREE="0  69/number 0/PARA 2/REGION 0/OCR"  CONTEXT="number null"></Start>
  <End  BEHAVIOR="Multivalent.Location"
     TREE="8  71/economic 0/PARA 2/REGION 0/OCR"  CONTEXT="economic null"></End>
</Span>

<Span  BEHAVIOR="Multivalent.std.span.BIUSpan"  CREATEDAT="Mon Feb 02 16:25:19
PST 1998"
     TYPE="Italicize"  NB="COPYEDNB">
  <Start  BEHAVIOR="Multivalent.Location"
     TREE="0  2/belatedly 0/PARA 2/REGION 0/OCR"  CONTEXT="belatedly have
come"></Start>
  <End  BEHAVIOR="Multivalent.Location"
     TREE="9  2/belatedly 0/PARA 2/REGION 0/OCR"  CONTEXT="belatedly have
come"></End>
</Span>
</Page>


<Page  BEHAVIOR="Multivalent.Layer"  NAME="Personal"  URL="inline"  NUMBER=4>

<Note  BEHAVIOR="Multivalent.Note"  NAME="NOTE2071152965"  X="200"  Y="200"
WIDTH="300"  HEIGHT="200"  POSTED>Hi Hal, \n \nBlah, blah. \n \nRW</Note>

<Span  BEHAVIOR="Multivalent.std.span.InsertSpan"  CREATEDAT="Sat Mar 07
09:06:38 PST 1998"
     ROOT="NOTE2071152965"  NB="COPYEDNB"  INSERT="how+insightful%21">
  <Start  BEHAVIOR="Multivalent.Location"
     TREE="0  0/Blah, 2/LINE 0/NOTE"  CONTEXT="Blah,  blah."></Start>
  <End  BEHAVIOR="Multivalent.Location"
```

```
     TREE="5  0/Blah, 2/LINE 0/NOTE"  CONTEXT="Blah,  blah."></End>
</Span>

</Page>
</Pages>


<Note  BEHAVIOR="Multivalent.Note"  NAME="NOTE1971181700"  ACTIVE="off"
BACKGROUND="green"  X="200"  Y="50"  WIDTH="300"  HEIGHT="68">Hub document
automatically generated \nSun Feb 01 12:35:21 PST 1998</Note>

</MULTIVALENT>
```

The toplevel tag, MULTIVALENT, maintains global properties particular to this document. (Properties available to all documents are stored in a user's individual preferences file.) The document's genre is examined for a cascaded hub that will bring to the fully instantiated document behaviors that are common to all documents of type Xdoc, behaviors such as the Show OCR and Show Image lenses (implicitly referenced is the hub with behaviors common to all documents of any genre). The behavior that manages paging to the next or previous or arbitrary page is informed to start on page 1 and limit movement to 9 pages.

The first behavior listed refers to the media adaptor for the Xdoc subtype of scanned page images. It provides two URL patterns that point to the image and OCR+geometry layers of the document, patterns that are instantiated with the current page number. Since OCR is run on relatively high resolution scans (600 dpi) but the images are shrunk to make them better suited for viewing on the screen and sending over the network, a SCALE factor is needed to convert original OCR units to screen units.

Pages wraps a annotations on individual pages. In its restore phase, Pages passes on restore to a child if and only if there is one whose NUMBER attribute matches the current page number. (This same technique could be used to generalize Pages to arbitrary tours of HTML pages, each with annotations particular to the tour.)

Page number 1 has a hyperlink span, an AwkSpan used to record a short comment, and a BIUSpan suggesting to italicize a span, all robustly anchored to the document as described in Section 5.2, "Robust Locations". Page number 4 has a Note with content "Hi Hal, Blah, blah. RW". Notes as full, recursive document trees, and name their roots so that annotations on Notes can refer to them, as is done by the following short comment annotation, via its ROOT attribute.

Outside of the Pages behavior and thus instantiated on every page is another Note, marking the date when a script dynamically constructed this hub.

4.5 General System Support

The system infrastructure described in this section is commonly found in digital document systems. The subsections below describe how these concepts are particularized to the Multivalent context.

4.5.1 Picking

Picking maps a given screen coordinate, often the cursor position, to the smallest containing element in the document tree. Picking is used in making the selection, clicking on hyperlinks, and elsewhere.

The screen coordinate is mapped to a node by first passing the coordinate through any the observers on the document root to acquire any coordinate transformations, such as from the Magnify lens (recall that lenses are implemented as observers on the document root). Then the coordinate adds the scroll position to make absolute document coordinates. A depth-first tree walk from the root translates to the relative coordinates at the current tree level by subtracting from the (initially) absolute coordinates of the offset of the current node. If an internal node contains the coordinate, it recurses over its children to see if any of them, which necessarily have bounding boxes no larger than their parent, contains the coordinate too. The first leaf to contain the point is returned as the mapping. If no child of an internal node contains the point, that internal node is returned as the mapping, which may be the root of the tree.

4.5.2 Grabs

When a behavior sets a grab, it receives all subsequent events until it releases the grab, bypassing the usual propagation though the document tree.

The default document manipulation bindings use grab to drag out the selection: When an otherwise unclaimed `MOUSE_DOWN` is seen over a document element, a selection is initiated; all mouse movement events are received directly, bypassing other potentially interested behaviors such as hyperlinks, until `MOUSE_UP`, which marks the endpoint of the selection to the default bindings, at which point the grab is released. Similarly, lenses use grab during lens movement: A `MOUSE_DOWN` on a lens titlebar initiated a move; `MOUSE_MOVE` events are received directly, triggering repainting of the area of the old and new positions of the lens; `MOUSE_UP` results in releasing the grab.

4.5.3 Style Sheet

A style sheet sets graphical properties for a document element by virtue of its structural role, as compared to settings done by observer behaviors, lenses, and spans. Style sheets

are useful for setting and easily changing the appearance of entire documents or document collections. The Multivalent style sheet accepts structure descriptions of the form:

> (*structural-element-name — parent-context — grandparent-context — …, graphical-properties*)

Presently there is no interpreted textual description of pattern-to-graphic properties mappings. Instead, the style sheet mechanism is programmed with the same general purpose programming language used in the rest of the system. Structural patterns are paired with a behavior that sets attributes of the graphics context. For example, the HTML media adaptor establishes a fragment of its corresponding style sheet with the following method invocations:

```
cx = new GenericStructure();
cx.setSize(18); cx.setStyle(Font.BOLD);
styleSheet.put("H2", cx);

cx = new GenericStructure();
cx.setTopMargin(0);
styleSheet.put("P", "H1", cx);
styleSheet.put("P", "H2", cx);

cx = new GenericStructure();
cx.setLeftMargin(20); cx.setTopMargin(0);
styleSheet.put("OL", "OL", cx);
styleSheet.put("UL", "OL", cx);
```

Once configured, the style sheet accepts queries for the structural style that should be applied at a particular point in the tree, if any. When searching for a style, the most specific style is selected, that is, the style with the largest amount of structural context.

In the future, the style sheet will be able to parse textual descriptions such as Cascading Style Sheets (CSS) [Bos *et alia* 1996] (currently individual style patterns are set with Java code). And styles will be able to associate arbitrary behaviors with structural elements, so that outlining and vastly different formatting can compose with existing tree nodes rather than requiring a specialized node.

4.5.4 Global State

Although behaviors are largely encapsulated units, sometimes a behavior acts according to general document state (such as the page number) or a group of behaviors need to coordinate via shared state. From a software engineering perspective, use of global state, which fractures the useful system building technique of encapsulation, is considered seriously: Every access to global state takes a method call, looking quite different than general variable use, and therefore encourages the programmer to think mindfully about the decision to make a variable global.

Global state divides into three independent categories of name-value pairs: persistent strings, general objects, and shared behaviors. String attributes are automatically saved as attributes in the hub document's root MULTIVALENT tag. Strings are used in numerous ways, examples of which follow. The Pages behavior keeps the current page number (converted to a string), which Xdoc uses to restore the proper page image and OCR data (see Section 6.1.3). General document meta information such as author, title, and other standard bibliographic data is used by the Provenance behavior to annotate pasted text with its source (see Section 2.3.6). Various display-related behaviors store the current display state, which is used both at runtime to coordinate among behaviors that can change or respond to this state, but also, since this state is persistent, whether to, say, show the document in Executive Summary mode when it is next loaded. Finally, during system development, profiling statistics are kept here so that actual system use can be tracked (and hence be made more efficient) over the lifetime of a document.

General objects can be kept in a separate global namespace. Notes, each with an individual, full-featured document tree, store mappings between their names and the root of their associated tree, so that when annotations are stored with reference to the Note name, they can when restored find the appropriate document root. Intra-document anchors associate the name of the anchor with the corresponding node to quickly jump to the anchor position without searching the entire tree for it. HTML stores client-side image maps, which can be used by multiple images. The URL of the current document is available for HTML to resolve relative URL's. The list of behaviors loaded in the system is used by the behavior editor to present a list of behaviors available on the network that have not already been loaded.

Shared behaviors coordinate behaviors actively, as opposed to passive shared data. Lenses can be implemented as ordinary behaviors with a coordinating controller managing lens overlapping (recall that lenses are not independent: where they overlap, their effects combine). The dimension of time was successfully introduced into the Multivalent framework with a shared behavior—without modifying the time-ignorant core of the framework. The experiment with time-based media [Matusik 1998] synchronized video, audio, subtitles, and general spans, which are made time-aware without modification.

For each type of state—strings, objects, behaviors—the sharing arises spontaneously and on demand as needed by behaviors; no specific accommodation is made by the core infrastructure. If the coordination group is not needed, its shared entity is not created. The standard pattern of interaction with shared state first tests for the existence of an object with the given name of the right type and creates this object if necessary. Usually this occurs with other initialization in the behavior's Restore protocol. For behaviors, which have a stereotyped instantiation mechanism, behaviors can use the convenience method called `getSharedBehavior`, which will return the shared behavior with the given name if it exists, and if not will instantiate it and return that.

Other system functions, including the Selection, could be implemented with the global state mechanism described here. Spans could be implemented without support built into

the core for their summary information kept on internal nodes. In fact, global state itself, with its three distinct types, could be implemented as a data structure kept as an attribute on the document root. But these functions were given their own interfaces, and guaranteed to be present thus obviating the need for an existence test, in order to provide a higher level interface to near-ubiquitous use.

4.5.5 Editing

An editing interface for inserting and deleting spans of document content is not a necessary part of the Multivalent framework. But the task is so common and so complicated (with spans on leafs, span summaries on internal nodes, linked graphical elements, editing in the middle of leaves) that a high level interface considerably aids programmers. (Spans themselves have a high level interface for creation, deletion, and movement.)

Insertion is more straightforward than deletion. Inserted media automatically acquire the prevailing spans over that region of the document. Insertion of full leaves of whatever medium is a simple addition to the tree, as this does not disturb the attachment of spans to existing leaves. Insertion of text into an existing leaf is slightly more complicated. After replacing the existing text of the leaf with the result of the insertion, those spans attached at the point of insertion or later are moved down on the same leaf by the size of the inserted text.

Deletion is more complex as it can cross structural boundaries and only one of the endpoints of a span. An initial pass over the region to be delete accumulates. Spans transition points. Spans that both start and end in the region are deleted; spans that either start or end but not both are recorded; spans that complete cover the deleted region are not affected. Next the leaves in the region and leaf parts (at the start and end of the region) are deleted. If all of an internal node's children are deleted, so it that node, recursively up the tree. Recorded spans are adjusted by moving the start of spans that started in the deleted region to the end; that ended in the region to the start.

5 Implementation: Performance, Scalability, Robustness

Realizing the Multivalent Document model as a practical system demands careful attention to data structures and algorithms to achieve interactive performance, scalability to large documents, and robustness in the face of change.

Performance and scalability are largely achieve through incremental, often hierarchical algorithms throughout the system. Rather than require a wholesale preprocessing for every change to the document, incremental algorithms are able to bring document data structures up to date with a usually relatively small amount of work, and if a large amount of work is required, these algorithms perform the work in small, interruptable units so as not to disturb the user experience. It is now believed that scalability of the system is less a technical concern that a social consideration of educating behavior writers to observe the principles of cooperation.

Layers are aligned in registration with one another and in practice layers can mutate independently of one another. Surveying the rate of change on the World Wide Web and considering the application to web page annotation, it is presumed that such mutations and resulting loss of registration will not be rare. Thus a practical Multivalent system needs to supply an elastic and readily available mechanism, robust locations, to record and reattach these points of registration, one that attempts to rederive the desired points in the face of mutation.

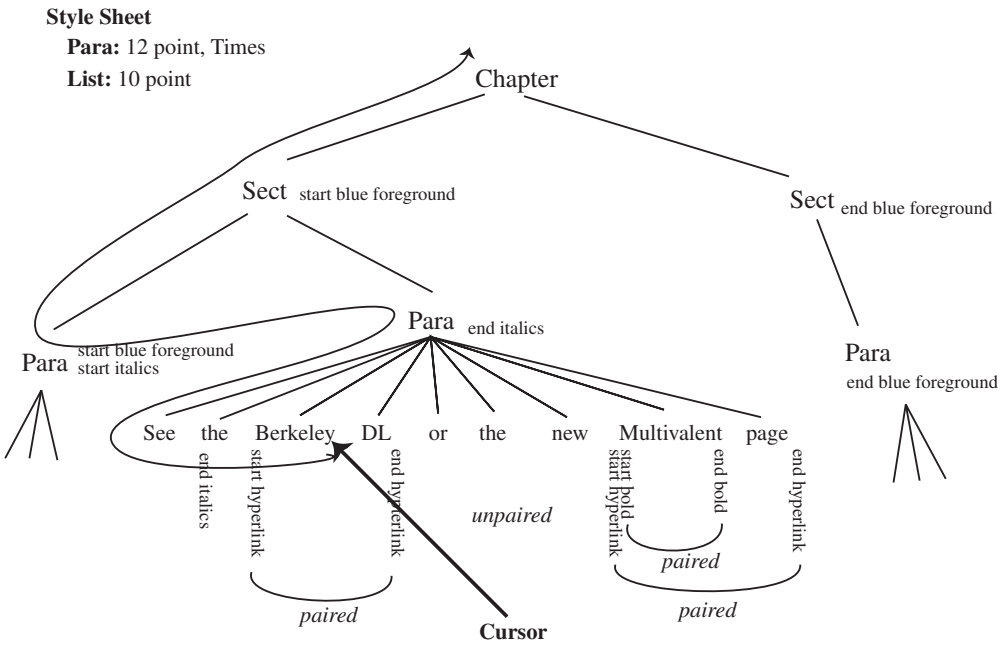
5.1 Performance and Scalability through Pervasive Incrementality

A rule of thumb states that as computer get faster, they are not used to solve the same problems in less time, but instead users spend the same amount of time solving larger problems. To maintain interactive performance as documents grow larger and more complex, it is important to use efficient algorithms. Often such algorithms are incremental, which in turn are often hierarchical. In the Multivalent implementation, hierarchical, tree-based algorithms often achieve a running time of O(log n), where n is some measure of the length of the document, and log n grows slowly with increasing n. Furthermore, incrementality must be pervasive, with one incremental algorithm flowing into another, or otherwise the one or more batch algorithms will become bottlenecks resulting in noticeable and annoying pauses for the user.

Some digital document systems employ various complex tactics to improve performance. As an example, Lilac [Brooks 1982], in part due to the lack of computing power available to it in 1982, even recycled pieces of the screen image. In the Multivalent Model, performance concerns were not allowed to interfere with the simple document tree that is the nexus of cooperation among behaviors. Even if, for example, a balanced binary tree would yield greater performance, the simple, logical structure of the tree was maintained, at least through some interface, and some other way to achieve the desired performance was achieved.

5.1.1 Efficient Computation of Active Behaviors at Given Point in Tree

Numerous operations critically rely upon rapid computation of active behaviors at a given tree node in order to achieve good performance. For example, given the following fragment of a document tree, assume that the user has moved the cursor over the painted area corresponding to the first hyperlink, precisely after the “y” in “Berkeley”. A general system service maps the cursor’s (x,y) position to the proper leaf and medium-specific position within the leaf (see “Picking”, Section 4.5.1). The algorithm computes the active behaviors at that point by following the indicated path.



Display: *See* the Berkeley DL or the new **Multivalent** page *(in blue foreground)*

Actives at cursor: hyperlink, blue foreground, 12 point Times

All of the incremental operations described below require an up-to-date graphics context so that formatting and painting are valid. Moreover, in order to synthesize enter and leave events for spans (so that, for example, hyperlinks can change the cursor when it is over a link), the system must map from a screen position to the corresponding tree node and from the node to the active behaviors; as behaviors enter the active set, the system synthesizes an Enter event; as they leave the set, a Leave event is synthesized. Since this happens continuously as the mouse moves about, computation of the active set is performed frequently.

Not only must the active set be computed rapidly, but any performance-enhancing auxiliary data structure should require storage proportional to the number of behaviors rather than proportional to the size of the document in order to scale to large documents, and it should be lazy or rapidly updateable as behaviors frequently change their scope.

A number of types of behaviors may be simultaneously active on a given node or position within a node. As detailed on Section 3.1, “Behaviors”, structural annotations and style sheets affect entire subtrees, spans cover leaves continuously from a start leaf to an end leaf, and lenses affect geometric portions of the screen. Structural annotations introduced by style sheets can change when the style sheet changes, spans are regularly added and removed as text is edited or annotated and can cover a little as a one unit-sized portion of a leaf to the entire document, and lenses affect different areas as they are dragged about the screen. All of this activity must be accounted for in computing active behaviors at a given point.

Algorithm

The implementation of a fast, lean algorithm to compute active behaviors is based on the Tk toolkit's text widget [Ousterhout 1993, Welch and Uhler 1996], which manages spans only. It is similar to "sticky pointers" [Fischer and Ladner 1979], but the premises of that algorithm leads to nonstructural mutations of the document tree, which is explicitly forbidden in the Multivalent model.

The algorithm combines information from various sources. Structural behaviors are suitably represented as is, as observers on the single node at the root of the subtree they affect. The algorithm does not keep auxiliary information, but rather simply takes structural behaviors into account during computation. Style sheets do not store anything in the tree, but like structural behaviors are taken into account during computation. Spans have a start and end unrelated to structure. The algorithm maintains auxiliary, hierarchical summary information about them in the tree itself. Lenses combine this information with an efficient top-down lens intersection algorithm (see Section 3.1.4). In short, only spans require an auxiliary data structure, and all three use some degree of on-the-fly computation.

The algorithm's auxiliary data structure for spans is built as follows. Given the start and end nodes, the algorithm computes the lowest common ancestor node. No action is taken to this or higher nodes in the tree. For spans that cover the entire document, this node is the root of the document tree, and for spans that start and end on the same node, no further action is taken. On the path from the parent of the leaf on which the node starts to the node immediately below the common ancestor, each node adds that span behavior to its list of spans that originate in its subtree. Likewise, on the path from the parent of the leaf on which the node ends to the node immediately below the common ancestor, each node adds that span behavior to its list of spans that end in its subtree. Thus, each node summarizes unbalanced spans in its subtree. (These lists could be kept as general attributes in the model, but spans are sufficiently pervasive to receive dedicated fields in the system.)

Computing active behaviors at a point within a leaf proceeds as follows. At the leaf, span begin and span end marks at earlier positions on the leaf initialize the active list. Span state within the immediate subtree is fully computed by traversing previous leaf siblings, which may or may not be leaves themselves, and collecting and keeping running totals of behavior start and end lists. When a behavior start meets its end, it is removed from the (start) list. Having summarized the immediate subtree, any structural behaviors explicitly associated with the parent are added to the start list (they will never meet an matching end mark), followed by any implicit behaviors associate with the parent via a style sheet. The subtree rooted at the parent has already been summarized, so its summary information is ignored. To summarize the parent's subtree, the parent's previous siblings are processed just as the leaf's was, and then the process repeats with the parent's parent. This process repeats until it reaches the root node. The list of active behaviors at the initial node is held by the started-without-end-seen list. (The ended-without-start-seen list is always empty at the end of the process as the algorithm examines a summary of the

entire tree left of the initial node, at which the corresponding start for every span in the end list must be found.) This algorithm is restated in the following pseudocode.

```
set list of started-without-end-seen behaviors to null
set list of ended-without-start-seen behaviors to null

while not at root
   traverse from node's immediate predecessor to first sibling in same subtree
       at each node
              append node's started-without to started-without list
              append node's ended-without to ended-without list
              if any behaviors found in both started and ended lists
                     remove from both

   set node to parent
   add structural behaviors found at parent to start-without list
   pass node name and positional pattern in tree to style sheet
       add matching behavior, if any

at root, add lenses to started-without-end-seen

return list of started-without-end-seen behaviors
```

The algorithm is fast. It is O(bh), where b is the branching factor of the tree and h its height, and the height is usually proportional to the logarithm of the document's size in nodes. It examines at most b nodes at each level of the tree (b/2 on average), at h (all) levels of the tree. The algorithm is space efficient. Structural behaviors, style sheet-induced behaviors, and lenses use no additional space. Spans, in the worst case, when they cover the entire tree, use 2(h-2) additional storage units, (h-2) as it excludes the leaf itself and the root, with (h-2) to summarize the start of the span from the left to the node immediately below the root, and another (h-2) to summarize the end. If a span covers less area, it stops adding start and end summarizers at nodes immediately below the common ancestor of start and end. Spans can be added and removed in O(h) time, to update the 2(h-1) auxiliary storage units. Moving a span is equivalent to removing and adding one.

Example

For the diagram shown in the previous section, active behaviors are computed as follows. First within the leaf, tracking leftward, the algorithm checks for associated behaviors. It finds a hyperlink mark point (which may be either its beginning or ending point), which is queried to determine it is a span start, and so adds it to the list of started behaviors. It traverses "the", finding no behaviors, and "See", finding an end italics span, which is added to the end list. Traversing up to the parent node named "Para" (the second Para in the diagram, reading left to right). This node has no structural behaviors attached, but does have a style sheet behavior that matches the name "Para", so it is added to the started list (without possibility of finding an end). Ignoring the summary at this node, as we have already computed the summary at a particular point within the subtree, and moving on to its left siblings, we find another Para. Since this Para is not on the direct path from the initial node to the root, we don't examine its style sheet. It has no

structural behaviors, but it does have start blue foreground and start italics spans, which are added to the start list. Both start and end lists have italics behaviors, so it is removed from both lists. Traversing up to Sect, it has no structural behaviors, style sheet matches, or left siblings. Traversing up to Chapter, the same holds as for Sect. Hence the active list, which is the final start list, is: hyperlink, blue foreground, 12 point Times, and the end list is empty, as guaranteed.

5.1.2 Incremental Formatting for Editing and Fast Startup

During document editing, many actions can result in changes to the document appearance: insertion or deletion of material (text, images, applets, and so on), addition or deletion or changes to the appearance of spans, and edits to style sheets. In another case, when reading a long document over a slow network connection, it is desirable to display document fragments as they become available so that one can begin reading quickly, before the entire document has arrived. This is important for HTML documents with many images, which individually can be much larger than the text of the document. Instead of reformatting the entire document for each character inserted or document fragment read in, an incremental algorithm is needed that recycles document appearance that is unaffected by the change.

The incremental formatting algorithm relies on a flag at every node in the document tree, a bit that indicates whether the entire subtree rooted at the node is fully formatted ("valid") or at least some part is not. For leaves, the bit indicates whether the corresponding adaptor for the media element needs to be asked for the leaf's dimensions. When some action invalidates part of the tree, the corresponding leaves are marked directly as well as all of their parents in a direct path to the root. When an portion of the tree is to be painted, it is first checked for validity, and if it is invalid, it is formatted on demand. Invalid nodes can have valid children, and if so, they are reused without reformatting. In this way formatting is incremental. Invalid nodes can be updated incrementally with a background process that simply calls format on various subtrees; if subtrees are chosen bottom up, the amount of time spent formatting is likely to be small and therefore not interfere with foreground operation, while the overhead of these many reformattings is small due to the incrementality. Of course, when the tree is invalid, document dimensions are incorrect. This may be reflected by a scrollbar, but in fact the user can still scroll through the document normally, as the scrollbar is incrementally corrected just as the document is incrementally formatted as more of it is displayed and formatted on demand.

While repairing invalid nodes, their size may change (due or inserted or deleted material or font changes), requiring reformatting its neighbors. When a change in a node's bounding box, which is examined at the end of formatting a that node, is detected, the node's parent is marked invalid and is immediately and directly reformatted. This process cascades until a node's bounding box remains unchanged or when the root is reformatted.

Of course, in reformatting a node an up-to-date graphics context is needed. The graphics context may need to be recomputed several times if various children are valid and there for skipped along with any changes they may have may to the graphics context. Thus this algorithm relies on the efficient computation of active behaviors at a given point in the tree that was described in the previous section.

This algorithm to format on demand the portion of the document in a given rectanglular window is restated below.

```
set list of started-without-end-seen behaviors to null
set list of ended-without-start-seen behaviors to null

procedure format(Node n, Rectangle clip) {
   if (n is invalid)
        save old bounding box dimensions

        if (n is a leaf)
               ask media adaptor for leaf's dimensions
        else /* n is an internal node */
               foreach (child of n)
                      format(child)
                      if (child's bounding box exceeds clip) break

        mark n as valid

        if (new bounding box dimensions changed from old)
               mark parent invalid
               format(parent, clip)
}

format(root, clip)
```

With this algorithm established, editing behaviors simply mark the extent of their effects and repaint the screen, and the system quickly brings the document appearance up to date. To display a document piecewise as it is being loaded over the network, the document format's media adaptor need only call repaint at opportune times as well formed fragments are added to the document tree. (Some structural entities are intrinsically interruptable. For instance, some HTML tables compute relative column sizes based on the amount of content in each column. Since additional rows could affect this computation, it is best to wait until full tables are available. Other HTML tables, however, declare column sizes and therefore can be shown incrementally.)

Furthermore, a relative coordinate system is employed in order that a change in a node's size (some inserted words require additional lines in a paragraph, say) does not result in updating the coordinates of every element after it in the document. A node's coordinates are given relative to its parent's. In document presented as scrolls (as opposed to paginated), a change in an element's height only involves repositioning its subsequent siblings in the tree, and its parent's siblings and so on recursively up the tree. All material in a subtree of a sibling is automatically, implicitly reposition since as its coordinates are always computed on-the-fly relative to its parent.

5.1.3 Incremental Painting for Scrolling, the Selection, and Lenses

Painting the entire document from start to end is straightforward. In the absence of lenses, the initial property values of the graphics context are computed from the style sheet. As tree traversal descends into a subtree, the style sheet is consulted for relevant structural properties, which, if found, are added in priority order to the context's list of active behaviors, and the new properties are computed vis-à-vis the new priority ordering. Likewise, as document painting draws leaves, at every span transition, the corresponding span is added via insertion sort on priority or removed from the active list as appropriate, and the properties recomputed. Without regard to efficiency, lenses could be supported by first adding the (relatively high priority) lens behaviors to the list of active behaviors and redrawing the entire document as above, clipping it to the lens boundaries. Similarly, painting documents that extend beyond one screenfull or that have been scrolled could be drawn in full and clipped to the visible screen.

However, it is of critical importance for interactive responsiveness that painting paint more or less only the area to be viewed. Therefore, efficiency in painting requires, in the first place, efficient tree traversal to, as much as possible, only those areas to be drawn; and secondly, efficient computation of the initial graphics context at that position. This latter is yet more important that at first appears as lenses in general draw noncontiguous portions of a subtree—their geometric shapes cut across linebroken paragraph lines—and as they do not encounter span transitions in the reading order traversal of the tree not shown in the lens, must reinitialize the graphical properties frequently. As with incremental formatting, the graphics context is efficiently computed as described above.

When a portion of the screen needs to be redrawn due to scrolling, lens movement or resizing, the document's window becoming un-overlapped, and so on, the instigating party constructs a clipping rectangle that bounds the "damaged" region. Scrolls of less than a full screen copy the area still valid into the new position and report the rectangle for the region that need be filled with material previously offscreen. Lenses report a clipping region encompassing both old and new positions, the old to be restored to show what is "underneath", the new to be freshly covered by the lens; old and new often overlap considerably. A system's window manager reports rectangular portions of the window, extending from one or two edges, which in the case of a window de-iconification, is the entire window.

Although the current implementation of the document tree is somewhat unfortunate insofar as it conflates structure and layout, and so cannot support multiple views or layouts that depart greatly from strict hierarchical layout, it nevertheless supports the usual range of layout and its essential organization is valid for the more general case. As described in Section 4.2, "The Document Tree", internal tree nodes hold a bounding boxes of their children. Given the clipping region, the tree is traversed from the root down. At each internal node, the clipping rectangle is translated into the node's relative coordinate system and compared against each of its children. If there is no overlap, the entire subtree rooted at that child is excised from further traversal. Thus, in worst case, the process of finding all nodes to paint is O(bh), where b is the branching factor of the

tree, and h the tree's height, which height is often log n in the number of leaves in the tree. Nodes with strong geometric orderings can improve linear search time through the children; for example, the standard hboxes (which lay out children strictly horizontally) and vboxes (strictly vertically), which layout their children in monotonically ordered, non-overlapping horizontal or vertical vectors, respectively, can use binary search, for a running time of O(log b * log n). This same general method is used in "picking", that is mapping a mouse position into a document tree node; see Section 4.5.1.

5.1.4 Performance Measurements

This section tests the theoretical performance of the incremental algorithms against actual performance measurements. As compared to a full formatting or painting of the entire document, the incremental algorithms should take time proportional to the fraction of the entire document that they operate on, with only a small additional overhead. The numbers measured here accurately compare *relative* performance, full to incremental; the implementation has not been optimized with measurements from a fine-grained profiler, so absolute performance, which is what counts in the end, is bound to improve. Nevertheless, current absolute performance does feel responsive, if not as quite as fast as, say, most mainstream word processors.

From the concrete document formats presently supported, HTML was chosen as the object of evaluation, as scanned page images are too small at only a page at a time (documents are collections of individual pages), and ASCII too plain to thoroughly test the algorithms. Three documents are measured, one of small size, one medium, one large. It appears that quite a number of web pages, especially home pages, are of small size, one or two screenfulls, not requiring scrolling to view everything. The present implementation's memory usage is too profligate as yet to accommodate very large documents (100 pages or more), but it can handle documents of size greater than an order of magnitude larger than the small case. The representative HTML pages chosen without qualification beyond their size are the DARPA home page (small, one screenfull/8K bytes in file size; www.darpa.mil), the Salon home page of 7 December 1998 (medium, just over 4 screenfulls/31K; www.salon1999.com), and the transcript of Frank Halasz's closing plenary talk at the Hypertext '91 conference, "Seven Issues Revisited" (large, more than 18 screenfulls/67K; http://www.parc.xerox.com/spl/projects/halasz-keynote/transcript.html). Measurements were taken running within Symantec Visual Café 2.5a (and its just-in-time compiler) running on Windows NT 4 on an HP Vectra XU 6/200 with a Pentium processor running at 200 MHz.

The following sections measure the performance of a supporting algorithm, and the incremental formatting and incremental painting algorithms.

Efficient Computation of Active Behaviors at Given Point in Tree

Any time an incremental formatting or painting is to take place, the current graphics context and set of active behaviors must be computed. Thus, this algorithm plays a supporting role rather than having a counterpart in the full document case.

The table below shows the results of timing the computation of behaviors active at various points in the documents, from the first leaf, through the first leaf of each subtree of the document root, to the last leaf. Active behaviors over a portion of the document may include spans, lenses and structural behaviors.

	First Leaf	Intermediate Leaves	Last Leaf
Small Document	0.61 ms	0.34, 0.28, 0.47, 0.38, 0.47, 0.38, 0.42, 0.39, 0.50	0.75
Medium	0.67	0.02, 0.02, 0.40, 0.41	0.88
Large	0.47	0.27, 0.67, 0.64, 0.38, 0.38, 0.38, 0.39, 0.39, 1.00, 0.41, 0.41, 80 more between 0.42 and 1.28	1.48

Measurement in milliseconds, averaged over 1000 iterations.

Most measurements fall under half a millisecond, with the highest almost reaching one and a half millisecond, or in other words about 1000 of these computations can take place per second. Times for the last leaf rise slightly over the next-to-last leaf as the last leaf must make full traversals of sibling nodes at each level to the root, whereas the next-to-last leaf is the first in the same subtree and so has no previous siblings at all but the next-to-highest level of the tree. Times did not rise appreciably from the small to medium to large document sizes.

Whereas one might expect an increasing progression from the first leaf to the last, what at first is perhaps surprising but is at second glance assuring is the uneven timings across the breadth of the document tree. Except for the last leaf as described above, each leaf is the first in its subtree starting at the root, meaning that subsequent leaves have only slightly more of the tree to traverse, just previous siblings just before the root. Instead, the timing variation is due to the amount of activity in the document, adding and removing behaviors from the active set during the tree walk. A behavior whose extant is entirely bounded within a previous subtree is not reflected in the summary node higher in the tree hierarchy, thus not contributing to the amount of work required during the tree walk and accounting for the periodic dips in time required.

Incremental Formatting for Editing and Fast Startup

Given an edit at a leaf, such as inserting or deleting a letter in a word, incremental reformatting first reformats the word. If the result requires the same amount of screen real estate (no more, no less), incremental formatting is done. If it does not, the algorithm chains to the parent and reformats it. The other children of the parent still have valid dimensions, thus all the parent need do is reposition them, accounting for the new dimensions of the edited leaf. If the parent's new dimensions match its old dimensions,

reformatting is done. If not, the process chains again, potentially up to the root. When reformatting is done, the affected area is repainted (incrementally, as described below).

The table below compares a full formatting with various partial formattings. Exactly what is reformatted in a partial formatting is highly data dependent. A single character inserted, deleted or changed may cause reformatting of the leaf node only, reformatting of its parent also, or potentially reformatting of the remainder of the document. Thus, the data for subtrees in the table selects various node types (with the indicated number of immediate children) and shows the time required for a reformat of each leaf in the subtree rooted at that node as compared to a maximally incremental reformat, with each child retaining a valid formatting, in which case the children need only be repositioned.

	Full Reformat	**Subtree HTML Node Type**	**Child Count**	**Subtree full reformat**	**Subtree incremental reformat**
Small Document	456ms (452 nodes in tree)	P(aragraph) BLOCKQUOTE	63 1	24.5 ms 11	2.5 ms 1
Medium	1045 ms (1387 nodes)	DIV(ivision) DIV P	1 3 46	939 29 21.7	2.35 2.66 2.3
Large	4690 (11444 nodes)	CENTER P (354)	1 354	5.78 205	1.09 9.2

All measurements in milliseconds; full reformat measurements averaged over 10 iterations, internal node measurements averaged over 100 iterations.

The times to fully format a document imply that it takes a bit less than half a second per screenfull (with the current unoptimized implementation). To support a typing speed of 100 words per minute, the algorithm must accommodate on average one character every 120ms, inclusive of repainting. Thus, even for small documents, some kind of incremental reformatting is necessary.

The incremental performance numbers suggest an order of magnitude speedup when the subtree is composed of the immediate children of the node (which they usually are for Paragraphs) and much more when the node is higher in the hierarchy (as for DIV, with a 400x speedup at one point). Depending on the extent of the ripple caused by the particular edit, the actual reformatting and redisplay cost is the sum of incremental costs for nodes from the leaf up to the node where the ripple ceases, plus the cost of the incremental painting for that final node.

Incremental Painting for Scrolling, the Selection, and Lenses

Various clients repaint stylized regions. Upon first reading the document or uniconifying the window containing the document, the full screen is drawn. The full screen is itself an incremental operation, as the screen is usually but a fraction of the entire document, which may be many screens long. Scrolling page-by-page draws most of a screenfull. Drag-scrolling, however, which requires good performance to feel responsive to the user, introduces a small amount of new document at the top or bottom of the screen; the rest of the screen can be bit copied to its new location. As it is being dragged to its full extent, the selection is shown by redrawing the smallest enclosing structural subtree containing it. Finally, lenses draw rectangular subportions of the screen, and require good performance when dragged about the screen to feel responsive to the user.

The table below displays measurements taken at the first and last screen of the document (upper and lower lines within each cell), and, as applicable, at several locations within that screen (numbers separated by commas within line).

	Full screen	**Drag-scroll: Up, down**	**Increasingly larger subtree for selection**	**Lens: top-left, middle, bottom-right**
Small Document				
first screen	429 ms	7.8, 1.7	4, 7	32, 143, 53
last screen	507	15, 9	4, 48	129, 134, 60
Medium				
first screen	217	9, 14	4, 90, 421	15, 28, 45
last screen	215	20, 12	7, 50	53, 54, 59
Large				
first screen	346	31, 84	51, 31, 57	100, 43, 162
last screen	329	46, 9	9, 3	57, 67, 50

Measurements in milliseconds, averaged over 10 iterations.

In painting the full screen, the time required does not depend on the overall length of the document or the position within the document, but rather with the amount of drawing to be done on that page, as indicated by the comparable times for drawing a screen across document sizes.

The measurements for scrolling are pairs for the time required to draw a rectangular portion of the document at full screen width and 30 pixels high at the top and bottom of the screen, respectively, exclusive of the time required to bit copy the remainder of the screen. These times, while dependent on the content of the document in that area, generally are an order of magnitude faster than drawing the full screen, and experientially speaking, make the difference between intermittent, jumpy drag scrolling and a smooth experience for the user.

The selection may increase from a single character to nearly the whole screen or more. The performance data indicate that repainting time generally scales with the amount of the screen to redraw. In a couple cases, however, the amount of time *decreases*. The

incremental algorithm imposes some amount of overhead to determine if various subportions of the document tree need to be drawn or not, and when subportions are omitted, recomputing the graphics context as may be missing modifications from these omitted subportions. In some cases this overhead takes more time than a brute force repainting of the entire area. Moving up to a larger region sometimes includes the former brute force region within its domain.

Lens drawing time depends on the size of the lens (here standardized at a medium-large 300x200 pixels) and the amount and complexity of document content in the lens. Incremental painting for lenses often obtains an order of magnitude speedup over drawing the full screen, declining to a factor of 3 in some cases and a factor of 2 in one.

In summary, the incremental algorithms do perform as desired, yielding performance essential for a smooth user experience.

5.1.5 Remaining Performance Issues

Several performance issues remain, for the most part due to adopting Java at an early point in its maturation and running in the highly constricted environment of an applet within a web browser. First of all, Java is slower than C, even with so-called just-in-time (JIT) compilers that translate Java bytecodes, which would otherwise be interpreted, into native machine code.

When moving from one document to another, the entire user interface is rebuilt. While the part of the interface that remains constant between the documents could be retained, a full rebuild should not be expensive. However, early versions of Java relied on so-called "native" user interface widgets, which apparently are expensive to instantiate and destroy. Current versions of Java have pure Java widgets, which are faster; alternatively, user interface widgets could be implemented as behaviors.

Presently the system relies on a proxy server to translate the error-filled HTML typically found in practice into balanced, well-nested tags. This imposes the overhead of a network hop, process creation, and a document parse. A native Java HTML corrective parser would potentially be faster.

Until recently, applets could not write to the local disk and it is still the case that they are given a fixed amount of memory. This makes it awkward to cache documents. This is one reason that future Multivalent versions will run outside of browsers, as independent applications.

5.2 Robust Locations

In the common, monolithlic data formats, all parts of a document are grouped in a unit. For instance, SGML mixes markup tags with the text they markup, and generations of the MPEG standard define ever more elaborate on-the-wire formats to accommodate more data types. When a monolithic file is edited, the editor can easily maintain registration between content (such as a word of text) and metadata (such as “bold”, “invisible”, or “section header”) since all affected parts are easily located and updated. In fact, many otherwise monolithic data formats have references to external data, typically for large data objects like images and video streams, and these references can break if references files are moved outside of the editor's awareness. Apart from this, monolithic data formats can guarantee perfect data registration.

In the Multivalent model, in contrast, the various subcomponents of a complex monolithic format are stored as individual layers. Layers may be stored in the same hub document, as separate files in the same directory, or as separate files distributed across the network. Moreover, additional data layers of possibly new data types can be bound into the single conceptual document. If all layers are always edited together, this would correspond to the monolithic model (with the advantage of easy addition of new data types), and the Multivalent model makes the same guarantee of perfect data registration.

In practice, however, the greater flexibility of Multivalent layers means that layers will be created and modified by various parties at various times, and thus layers are jilted out of sync. Because individual, possibly distributed layers of data is a fundamental and pervasive aspect of the model, it is especially important have a system to record in one layer a location in another robustly enough that the location can be redetermined in the face of change. Since many behaviors could and should use such a service, it has been included in the core system.

A standard, shared location module generates descriptions and resolves previously generated descriptions for locations within the document tree. (Descriptions can be extended by media-specific behaviors with additional attributes, such as frame number or SMTP timecode in video, bar or measure in a music score, or a simple byte offset in a raw audio stream.) A location descriptor redundantly records several types of positioning information, each with different strengths.

When a saved location description is resolved to a runtime document, if the document was unchanged since the description was generated, that same runtime location is guaranteed to be recovered. If the document has changed, the probability of correct reregistration is inversely proportional to the amount of the change; that is, there is no sharp drop in recovery capability. In the face of change, a variety of backoff tactics is applied within each descriptor subtype, and a failure within one subtype drops down to the next, in decreasing order of quality. Of course, at the extreme, if a location points to a word and that word’s entire chapter is deleted, that logical location no longer exists and cannot be repositioned, and so in this case, the system reports this fact with the associated, now obsolete positioning information to the user. (Behaviors that rely upon

locations, such as spans, are robust to reattachment failure.) Once attached, locations are maintained exactly during runtime editing, and updated locations descriptors are written out when the document is saved.

Locations cost about 100 bytes each. Spans have two, one for the start and another for the end points. In the future, if versioning information were guaranteed, smaller descriptors could be used and repositioning would be even more robust. Such versioning could be done either at the client, perhaps with some policy that annotated pages are permanently cached, or at the server.

Preliminary results with a prototype, less sophisticated implementation in TkMan indicate that in practice the annotation repositioning strategy works well; of 754 annotations that needed repositioning to layers that underwent varying degrees of mutation according to uncontrolled changes in man pages, 742 annotations were automatically repositioned, leaving 12 to be reapplied by the user. In most of the latter cases, the associated position had in fact been deleted entirely.

5.2.1 Location Types

The three subtypes in the standard location descriptor are unique id, tree path, and content with context. Behaviors can augment the standard descriptor with medium-specific information.

Other location types were considered. A simple byte offset is brittle—a simple insertion or deletion change immediately breaks all subsequent locations—but can be useful as an initial search point for recovering the closest match in some data formats. A byte offset is better suited for a streamed medium; since the document tree is hierarchical and is better served by a tree path, the byte offset is not included in the standard descriptor.

HyTime [ISO92], which defines SGML constructs for hypermedia and time-based media, provides three location types: unique id, position in general list (often of tree nodes TREELOC or IDs), and querying by property. Multivalent location types includes unique id. HyTime does not address location recovery strategies in changed documents. In particular, it does not record node name at each point along a tree walk, so a reattachment procedure cannot determine when the walk goes astray, except when a child number is requested that exceeds the number of available children. Querying is not appropriate as a persistent descriptor for a particular location: a unique location is desired, but queries find all matching a set of conditions, and it may not be possible to generate such a set to uniquely identify location, and even if it were, determining the set could be computationally expensive. HyTime does consider time-based media, however, so it may be useful to behavior writers looking to extend the standard Multivalent location types to these media.

Unique Identifiers

Unique identifiers (UIDs) are usually invulnerable to editing, unless the element itself is deleted, and so are preferred. But UIDs are often only available at key points in a document, and a correspondence between layers—a copyeditor annotation, say—can be attached anywhere. UIDs could be mechanically generated for every position, but this would only be possible for document formats that support them, SGML but not ASCII or scanned images, and only if the source document can be modified. In practice, UIDs are preferred when available, but one is often not available at the precise point of registration.

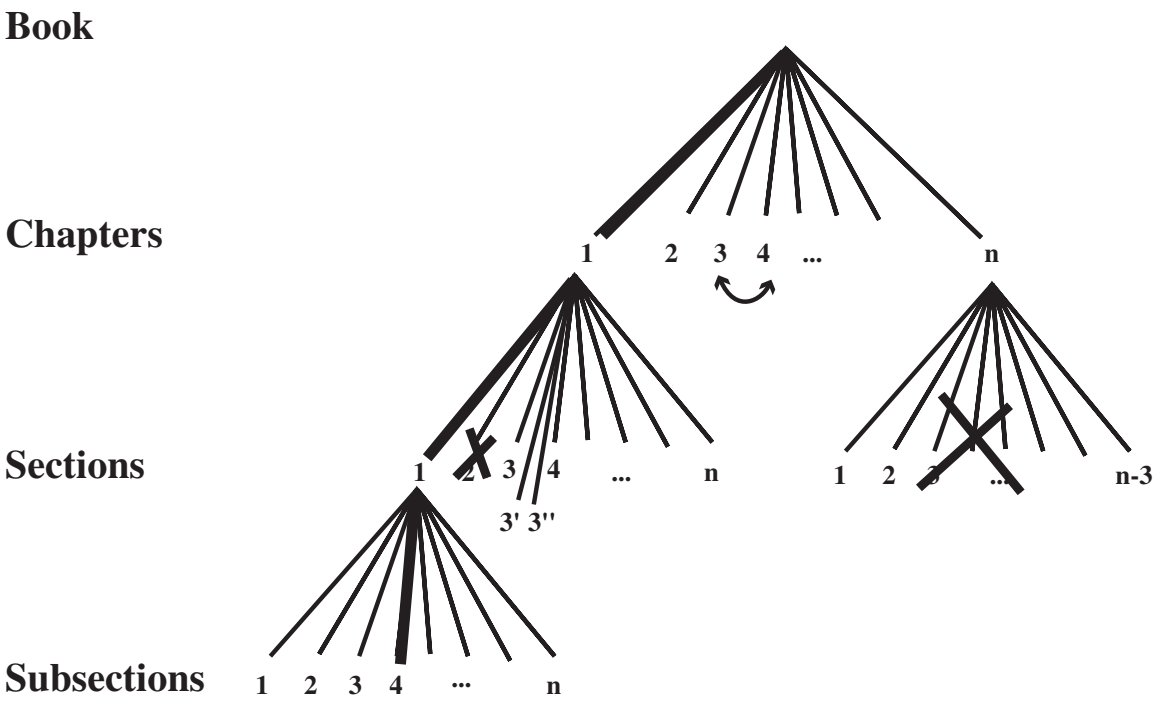
Tree Path

The tree path naturally corresponds to the document tree and exhibits strong natural robustness. The tree path is recorded from the offset into a leaf up to the root, with a tree node recorded as a child number within its parent and the node's name. For a text location within an HTML document, as a concrete example, node names corresponding to structural tags (H1 or LI, not spans such as B or TT), and the leaf name is a word, as in the example tree path below. Reading right to left, the example can be decoded as starting at the root named "HTML", taking the first child (number zero) to a node named "BODY", taking its first child to a node named "H3", and so on down to the leaf, and finally pointing to medium-specific position 8 within the leaf.

```
8  0/Professor 0/ADDRESS 1/H3 0/BODY 0/HTML
```

On rare occasion, performance considerations pressure a behavior to introduce a nonstructural level to the document tree (see Section 4.2.2). All such nonstructural nodes are given a null name, which the location module ignores both during descriptor generation and resolution.

Fortuitously, the simple tree path exhibits strong natural robustness. In resolving the path from root to leaf, changes of any magnitude within *following* sibling subtrees are safely contained. The diagram below depicts an abbreviated document tree, showing only selected nodes and no node labels. Assume the saved path describes a traversal down child 1 from the root, along child 1 of the chapter, and along child 4 of the section. As the diagram shows, this path is robust to the interchange of chapters 3 and 4, the deletion of section 2 of chapter 1, the insertion of two sections between numbers 3 and 4 in chapter 1, and the deletion of any number of sections from chapter n.



Changes in *previous* sibling trees along the path do cause tree paths strictly based on child offsets to fail. If the path should traverse child 4 at some point but children 1, 2, or 3 were deleted, or children were inserted before child 4, then the desired child's offset would no longer be 4. Thus node names are recorded as well to check the numerical child offset. If in trying to match a point on a saved path with the actual runtime tree fails at the exact saved offset, matching is tried on the following sibling node, then the previous sibling, then the second following, the second previous and so on in the pattern of displacements +1, -1, +2, -2, +3, -3, and so on until the name of the node names match and, for internal nodes, their remaining subtrees match as well. If no match can be found, attempts are made assuming that a level of hierarchy was introduced by skipping a level of the tree. If still no match is found, attempts are made assuming that a level of hierarchy was removed by skipping the current location node/offset descriptor pair. If none of these attempts yields a match, the match attempt fails. Each level of inexactness from the original description adds a weighted value to an overall "fuzziness" factor for the match thus far, and if the factor exceeds its bound, the match attempt fails.

It may happen that a numerical offset is wrong and the replacement node has exactly the same name, in which case a match would be reported at that point; however, it is likely that such a coincidence will not occur for the remainder of the subtree rooted at that node, and this false match will be caught, and another match attempted. A failure at a node is reported to its parent, which will cause it to search increasingly distant siblings.

This algorithm makes saved tree paths robust against any number and combination of sibling insertions, deletions and reorderings at any or numerous levels of the document tree. The search pattern prefers siblings closest to the saved location, rather than, say, searching from the parent's first child to its last, reasoning that the closest match is

probabilistically most likely the best match. This search tactic has the desirable algorithmic property that the first match found is also the closest.

Content and Context

If the structural document tree has been changed too much for the tree path's tactics to recover—perhaps a sentence has been cut from one chapter and pasted into another—an attempt at re-registration is attempted using the content and context. Strictly speaking, locations are points and have no content, and everything is context; for the purposes of this discussion, however, content is defined as the leaf tree node within which the location is found.

As with the tree path, the closest match to the original position is the preferred. The questions for systems that use content and context are where to start searching, what is the nature of the content and context, and backoff tactics in the face of change. ComMentor [Roscheisen *et alia* 1995], which annotates HTML pages, records the shortest unique substring (based on string position trees, implemented as Patricia trees) and redundant context. As long as the string is unique, initial search is unimportant. The UNIX utility patch [Wall], which applies file "diff"s, or differences between files to make one identical to the other, records line numbers of the line with the difference and some lines of context before and after. It starts searching from the position of the last patch plus the saved offset between patches, and searches forward and backward in the file for a match of all lines, including context; if no match is found, it drops more and more lines of context. The prototype for Multivalent implementation, TkMan's [Phelps 1994] highlight annotation, uses line number and character offset within the line as the initial search position, and records content and context at start and end of the highlight span. A search is done forward and backward and the nearer match chosen. If neither direction matches, more and more of the context is shed until one does or their context drops below a threshold. If the match is beyond a threshold distance from the initial position, the user is asked to verify the repositioned highlights. Failed matches are reported at the end of the page, as with ComMentor.

The initial search position is set from the furthest extent that the tree path described above could be resolved. This sets a restricted domain within which to search, which benefits proximity matching and performance. If no match is found within a subtree, the search climbs up the tree by one node and tries again until a match is found or it has climbed to the root. Within a subtree, leaves are searched left to right. A match must match the content and context.

Another level of backoff leniency could allow inexact, or fuzzy, matches, that is, a small number of differences as computed as minimum string length difference. In the case of patch, with its line-size level of granularity, this would likely be more valuable than in other systems with character-size levels, where this extra step is much further into the law of diminishing returns.

As with tree path matching, an overall fuzziness factor for the match is maintained. As the search examines ever larger subtrees, the fuzziness is increased. If some but not all of the context is matched, fuzziness is increased. If fuzziness exceeds a threshold, the search fails. Empirical experiments are needed to determine good values for various types of fuzziness in order to obtain the optimal balance between missed positive hits and false negatives.

5.2.2 Comparison

The following table summarizes the types of robustness in locations.

Method	**Robust to change type**	**Backoff tactic**	**Quality of reattachment**
UID	any except deletion	n/a (perfect recovery)	Perfect
	Deletion	none	n/a
tree path	insert/delete/reorder siblings	search siblings of decreasing propinquity	Gracefully degrades
	add/remove level of hierarchy	skip or pseudo match level	
Context and content	location moved anywhere	search within matched tree path, prefer closest	Gracefully degrades
	move and c and/or c changed	fuzziness +3 or content, +1 for each size of context	
	move and c a/o c	fuzzy match (unimplemented)	

6 Applications

Preceding chapters have motivated the need for a radically extensible digital document model and described an architecture for its realization. Yet, the infrastructure performs no useful task as far as the end user is concerned. All user-level functionality is implemented as extensions, as behaviors that provide a user interface and translate the user's wishes into data structure manipulations and communication with other behaviors.

In this chapter, we examine three applications for the infrastructure: enlivening scanned page images, extensible HTML, and distributed annotations in situ. Application to ASCII and UNIX manual pages were described as concrete examples of the Multivalent protocols (see Section 2.3). These first two applications define a continuum of concrete document formats. Scanned pages images are pure appearance; all semantic content is derived through analysis. HTML, ideally if not in practice, is pure semantic markup; its appearance is given entirely by some formatting engine, an appearance that is explicitly allowed to vary across engines. The annotations implementation leverages the layered document model and fine-grained extensibility of the model to achieve significant advantages as compared to other annotation systems.

While useful in their own right, the applications also support theoretical claims. Applications with unprecedented capabilities, implemented relatively easily support the claim that the Multivalent model is highly extensible. Applications help validate the document model with proof that it is capable of supporting useful work. Applications help flesh out the document model for the implementor by starkly revealing where more work is needed to support desired functionality. The diversity of applications demonstrates the adequacy of the simple model. The breadth of applications show that the model is generalized so that a single behavior implementation can operate without modification across the continuum of types; and yet rigorously austere, barring any special accommodation for a particular application from creeping into the core.

This chapter first motivates each application, then shows the Multivalent solution and describes how the Multivalent protocols were specialized to implement the application, and finally considers work related to the specific application, as distinct to work related to the system as a whole.

6.1 Enlivening Scanned Page Images

6.1.1 Problem

We say "digital documents" to distinguish them from paper documents. Paper is ubiquitous and has many useful qualities. Paper is inexpensive, easily portable, and highly readable. On the other hand, large collections of digital documents can be quickly searched and they can be copied perfectly and instantaneously over networks. A digital facsimile of paper can be made by scanning it into a computer image. Optical character recognition (OCR) can analyze the image to extract the text, images, formatting, and other structures. Given the state of the art of OCR, few applications would want to discard the page image because the OCR translation is unreliable, introducing an increasing number of errors as the document varies from simple typewritten, single-column, single-font, illustration-free, table-free, English text.

Thus for scanned documents of any complexity one may prefer to work with the page images, for the image is the digital representation with maximum fidelity to the paper original. As the base for further digital manipulation, the page image is very likely to be available, and it has the fewest errors, if any, among digital representations. Working with a high fidelity representation is especially important for historically significant documents. "Historically significant" includes not only the rarified and internationally treasured works such as Gutenberg Bibles, but from the proper historical distance and for the right audience such documents can include the Talmud, legal papers, the New York Times from December 8, 1941, Aunt Charlotte's Birthday card, handwritten manuscripts, illuminated books, and fine press materials, where the typography materially contributes to the value of the work. This is illustrated in the quotation below for music composition:

> Printed music, like printed text, tells us rather little about the creative process. But composers' manuscripts are full of information, much more so than literary manuscripts, because musical notation is far more fluid, rich and diverse, both in its symbols and in its levels (melodic line, phrasing, dynamic markings, instrumentation). There is a huge repertory of signs by which a composer's momentary thoughts can be closely documented on the page.
>
> "What Manuscripts May Have to Say About Music",
> Paul Griffiths, New York Times, May 31, 1998, section 2, page 29

Even were the OCR were to be perfect, translations into other digital representation such as SGML inevitably lose information as they are not capable of expressing the entire information content that the human can read into the image. And it is difficult to predict what needs preserving; at the extreme, it is possible that in the course of time that even the worm holes become significant, as a variant interpolation of the text eaten out may yield a different translation, and, perhaps, a new interpretation.

Page images may be preferred for more mundane documents as well, such as handwritten papers, documents with mathematical or musical content, or even word-processed documents with handwritten comments. OCR and page analysis fail to analyze these types of documents well. Even if a document is ideal input for OCR, page images may yet be preferred if no errors can be tolerated, perhaps due to legal liability.

Scanned page images is representative of a class of "fixed format" documents including. PostScript, PDF, and TeX DVI., among others, where the final appearance of the page is described minutely. One purpose of this level of detail is to produce identical copies of a document on a variety of printers from a variety of computers, but our concern is online manipulation.

Yet while the image is in many ways preferred, to most systems, including standard web browsers, the image is a black box image little different from an image of a sunset; it is simply a pattern of bits that humans can assign meaning to, but that has no semantic information for the machine. In moving to the image as the primary visual representation, we have surrendered all the document manipulations, such as cut and paste, expected in word processors. In fact, one can consider scanned page images the most recalcitrant type of digital document because any treatment of the image as a document page must rely upon additional analysis not contained in the image itself.

6.1.2 Solution

The Multivalent manipulation of scanned page images can treat the images as documents by utilizing multiple representations of the page: image and OCR, a mapping between the two, and other analyses as available, such as table cells and math and bibliographic regions.

Recall from Introduction chapter (where several illustrative screen dumps are available) the functionality supported by the Multivalent framework with appropriate behaviors. Users can select text and paste the corresponding OCR into other applications. Selections flow from column to column or through other structural elements. The pasted text can be not just OCR but, where greater structural information is available, bibliographic entries given as BibTeX or other formats, mathematical given as Mathematica format or executable Lisp, and so on. Lenses can magnify the image or show the geometrically positioned corresponding OCR, or both if the lenses overlap. Pages can be annotated with highlights and hyperlinks; when annotated with copy editor markup that carries a

short message, the lines of the scanned page reformat slightly to open up screen space in which to display the message.

6.1.3 Implementation: Specializing the Multivalent Protocols

The user of the Multivalent scanned page browser works with the image, the highest fidelity digital representation of the original document. Description of the image as a semantically rich document is built up by an incrementally extensible set of layers: word (ASCII) transcription and geometric mapping in the XDOC format, table bounds, bibliographic entry structure, annotations, language translations an so on. Behaviors exploit this data to manipulate the appearance in useful ways.

The Multivalent scanned page application shares a high degree of code with every other application. Adapting the general infrastructure described in previous chapters to work with scanned pages involves a small—but well informed—amount of additional work. Typical of adaptation to other document formats (as opposed to other media types such as video), the customization for scanned images involves specializing (in object-oriented terminology, subclassing) a media adaptor and supporting tree nodes.

The specialization of the Multivalent extension for scanned pages entails only a media adaptor and supporting tree leaf (and it takes advantage of an internal node type that is aware of fixed format document types). This section considers the document protocols individually, detailing how each was specialized—if at all—to implement the various features of scanned pages.

Restore

A representative hub document for a multi-page scanned document lists the following behaviors and associated parameters below. Actual hubs often have many more behaviors, but the following does not oversimplify any didactic considerations.

Behavior	**Attributes**
Pages	Page=9
Xdoc	URL of images, URL of OCR
ShowOCR	
Highlight	Document location pair 1
Hyperlink	link destination, document location pair 2
Note	x,y, width, height, <textual content>
InsertCopyEd	Document location pair 3
Highlight	Document location pair 4
Biblio	<fielded bibliographic information keyed to location in document>
BibTeX	
BibRefer	
DefaultEvents	

The restore protocol first sets from the hub the global document attributes author, title, and source.

The Pages behavior is generally useful for documents with multiple pages (scanned pages, PDF), multiple screens (slide shows such as PowerPoint), or multiple points along a trail (guided tours). The system directly restores only Pages, which has nested within it all other behaviors. Pages sets a global attribute called PAGE to the current page/slide/point on tour and then recursively instantiates its children.

The specialization for scanned page images comes with their media adaptor, called Xdoc. OCR software from Xerox can produce a format, called XDOC, that not only transcribes the text from a scanned page image into ASCII, but also supplies geometric mapping information between the ASCII and their source region (a rectangle) in the image. (Other adaptors could be developed for other image-text mapping formats, hence the non-generic behavior name.) The initial portion of a sample XDOC file follows.

```
[a;"XDOC.9.0"]
[d;"00000008"]
[p;1;P;1;1428;0;0;2171;2804]
[s;1;439;0;151;c;1][e;1][c;1]DEPARIMENT[h;671;8]OF[h;726;8]WAThR[h;851;8]RESOUR
CES[h;1061;12]-
[h;1083;10]CALIFORNIA[h;1306;7]DATA[h;1404;5]EXCHANGE[h;1604;10]CENThR[y;1749;0
;151;1;H]
[s;1;439;161;187;c;1;6]ThLEMETERED[h;853;10]SNOW[h;965;8]WAThR[h;1091;8]EQUIVAL
ENTS[h;1336;12]-[h;1357;19]APRIL[h;1477;10]1,1990[y;1749;160;187;0;H]
[s;2;138;1274;250;p;2;88][e;2][c;2]INCHES[h;1514;8]OF[h;1558;7]WATER[h;1660;7]E
QUIVA[ET[y;1899;55;250;1;H]
[s;2;138;0;281;t;2;0;1]BASIN[h;223;6]NAME[h;312;529;36;1]ELEY[h;912;119;8;1]APR
[h;1086;9]1[h;1104;309;21;1]PERCENT[h;1538;66;4;1]24[h;1632;8]HRS[h;1694;104;7;
1]1[h;1807;10]wEEK[y;1899;0;281;1;H]
```

(many more lines follow for this scanned page)

Commands begin with a left bracket (‘[’) followed by a command letter, parameters separated by colons, and end with a right bracket (‘]’). In between commands are recognized words (thus location information is given at word-sized granularity). In the example above, the “a” command marks the “start of document” with a parameter giving the version of the XDOC specification to which to output was generated. The “p” command marks the “start of page”, with page number, orientation code (“P” for portrait, “L” for landscape), and the bounding box (coordinate 0,0 with width 2171, height 2804), among other parameters. The “s” command marks the “start of text line”, with the zone number (1, for the first “s” command, one the fourth line), y-coordinate of baseline (151), and identifier of primary font (0), among other parameters. The “h” command indicates nonprinting whitespace, with the x-coordinate of its start and its width.

The Xdoc media adaptor is aware of the PAGE attribute and uses it to complete its URL patterns to the individual page within a set of similarly named files for a document, and loads two supporting data layers for that page, the scanned page image and the corresponding augumented OCR, in XDOC format.

Build

Having read in the page image and XDOC in Restore, the Build Before method of the Xdoc media adaptor parses the XDOC and constructs a document tree. Ideally, the document tree is a hierarchy of semantic structure: sections enclosing subsections enclosing paragraphs. Raw XDOC reports physical properties only, and the most faithful tree hierarchy has regions enclosing heuristically derived paragraphs enclosing lines enclosing words.

Internal tree nodes are of a class shared among fixed-format documents such as PDF and DVI. This internal node type does not position its children, linebreaking or otherwise, but rather accepts their positions and computes its bounding box around them to maintain the nested bounding box invariant.

The leaves of the tree have been specialized for Xdoc with a class called IdegTextXdoc. For reasons given in Format and Paint below, this leaf type has two bounding boxes, one mapping to the original page image and a second mapping to the generated page image actually drawn for the user.

Generating the mapping from XDOC information requires processing beyond parsing. Most likely the OCR translation was made on a high resolution scan, typically 300 or 600 pixel per inch (ppi). However, the working page image has been scaled for the screen, to 75 or 100 ppi, so the coordinates need scaling. To enable its incremental properties, the Multivalent model mandates that child nodes be given coordinates relative to their parents. But the XDOC coordinates are absolute, and moreover not computable until font metric information has been read, which in some versions of the XDOC format comes at the end of the file.

In the system's Build After pass, annotations, including hyperlinks and copy editor markup, are attached to the document tree, according to generic Build After procedure.

Build is the last phase involving the Xdoc media adaptor. Subsequent phases rely on internal tree nodes designed for fixed-format media, including PDF and TeX DVI in addition to Xdoc. The new tree leaf class handles specialization for Xdoc in particular.

Format

In the absence of annotations, Format takes the absolute coordinates at the leaves and computes relative coordinates for all nodes in the tree, internal nodes included. This is an intrinsic part of the document as constructed and so this is done by tree nodes, specifically fixed-format aware internal tree nodes. Behaviors can modify the usual formatting, to implement outline views say, by hooking into the relevant nodes and taking action when the behavior's Format Before and Format After methods are invoked.

Copy editor annotations make greater demands on Format. Consider making a short copy editor remark referring to the middle of a line of single-spaced text. With paper, one might draw an arrow to the margin and write the comment there. We can take advantage

of the malleability of digital documents to place comments in situ, at the exact point they apply. This is accomplished by enlarging the effective height of the character at the point of comment, opening space in which to write the comment. Reformatting "fixed" text requires propagating the change in position to subsequent text—below and to the right—so that text does not overlap. The screen dump below to the left displays a portion of the original page; the right displays the document after annotation with two short comments. Notice that the page was reformatted just enough to make space for the comments, pushing text below down by just the amount of space required, and leaving the unaffected text in the next column undisturbed.

COOPERATING AGENCIES

Private Organizations
J.G. Boswell Company
Kaweah River Association
Kings River Water Association
St. Johns River Association
Tule River Association
U.S. Tungsten Corporation

Public Utilities
Pacific Gas and Electric Company
Southern California Edison Company

Municipalities
City of Bakersfield
Water Department
City of Los Angeles
Department of Water and Power
City and County of San Francisco
Hetch Hetchy Water and Power

State Agencies
California Department of Forestry
& Fire Protection
California Department of Water Resources

Federal Agencies
U.S. Department of Agriculture
Forest Service(14 National Forests)
Pacific Southwest Forest and Range
Experiment Station
Soil Conservation Service
U.S. Department of Commerce
National Weather Service
U.S. Department of Interior
Bureau of Reclamation
Geological Survey, Water Resources
Division
National Park Service(3 National Parks)
U.S. Department of Army
Corps of Engineers

Other Cooperative Programs
Nevada Cooperative Snow Surveys
Oregon Cooperative Snow Surveys

COOPERATING AGENCIES

Private Organizations
J.G. Boswell Company
Kaweah River Association
Kings => King's?
Kings River Water Association
St. Johns => St. John's ?
St. Johns River Association
Tule River Association
U.S. Tungsten Corporation

Public Utilities
Pacific Gas and Electric Company
Southern California Edison Company

Municipalities
City of Bakersfield
Water Department
City of Los Angeles
Department of Water and Power
City and County of San Francisco
Hetch Hetchy Water and Power

State Agencies
California Department of Forestry
& Fire Protection
California Department of Water Resources

Federal Agencies
U.S. Department of Agriculture
Forest Service(14 National Forests)
Pacific Southwest Forest and Range
Experiment Station
Soil Conservation Service
U.S. Department of Commerce
National Weather Service
U.S. Department of Interior
Bureau of Reclamation
Geological Survey, Water Resources
Division
National Park Service(3 National Parks)
U.S. Department of Army
Corps of Engineers

Other Cooperative Programs
Nevada Cooperative Snow Surveys
Oregon Cooperative Snow Surveys

Reformatting in this way is a delicate operation. First the coordinates for the entire tree are computed based on original XDOC coordinates without regard for adjustments. As annotations are added, the change in position is computed. Tree node siblings are scanned for would-be overlap (since deltas are positive, overlap can occur only with text below and to the right), and overlap is prevented by adding the amount of the change to the sibling's coordinates. As formatting progresses up the tree, when an internal node computes its bounding box, it is compared to its bounding box based on raw XDOC. If the bounding box size has changed, it adjusts for overlaps with its siblings. Thus, reformatting is efficient as it touches only nodes in the path from the change to the root and those nodes' siblings, rather than touching every node in the entire tree. And reformatting is minimally invasive, as when a subtree is moved to avoid overlapping, the relative position of nodes within that subtree are maintained.

Paint

Paint for scanned images can usefully render the word at that leaf as OCR or image. For ASCII, it merely sets the correct font and invokes Java's text drawing package. For the image, Paint utilizes both bounding boxes maintained by IdegTextXdoc. The first, with coordinates in terms of the original, is used to identify the proper source region within the original image, and the second places it at the current target location of that word, after possible reformatting. Rather than taking the full page image and chopping it into hundreds of word-sized images stored with each leaf, leaves share access to the full image kept by the Xdoc behavior. When IdegTextXdoc leaves are created, Xdoc sets the

owner field common to all nodes to itself, and leaves use this link to locate the full page image.

When a medium renders itself, it is obligated by Multivalent protocols to adhere to the current settings of the graphics context (see Section 4.3, "The Graphics Context") as far as possible for that medium. For scanned pages, it is infeasible to draw in the current font. The background color, used by selections and highlight annotations, is respected by first identifying the background color (the color closest to white) and making it transparent, to be supplied with the current color as that word is drawn. Due to inefficiencies in image support in earlier versions of Java, zooming, as for the Magnify lens, is implemented by brute force, but this will be revised to drawing words at the current zoom factor. The choice of whether to draw words as ASCII or image, used by the ShowOCR lens and the View Page as OCR menu option, is given by a medium-specific signal in the graphics context. The IdegTextXdoc leaf queries for the XDOC property; if defined to ASCII, it draws as ASCII, otherwise as image. Other media types ignore this signal harmlessly, for while they are unaware of its existence, neither do they have a choice of image or OCR appearance.

Events

Basic scanned page support does not specialize event support beyond reporting its word-size granularity as size 1, that is, either the entire word or nothing; the cursor can be before or after a word, but not in the middle. A media adaptor with character-granularity information would probably report granularity size equal to the number of characters in the word. Selections across words, lines and columns depend on the fact that Build constructed the tree with a reasonable order of text flow, which may have required geometrically sorting the sequence of XDOC regions.

Other behaviors that rely on events, such as hyperlinks on mouse clicks, make no special accommodation for nor receive any special accommodation from scanned pages.

Clipboard

As the system builds up the selection and requests the content of a IdegTextXdoc leaf, the leaf reports its OCR representation. Other behaviors involved with the clipboard, such as the generation of various concrete formats for selected bibliographic entries, make no special accommodation for nor receive any special accommodation from scanned pages.

Undo

Scanned pages themselves cannot be edited so Undo is not applicable.

Save

No specialization is made in the Save protocol.

6.1.4 Related Work

Image EMACS

Other systems have treated page images as documents. Image EMACS, "a text editor for scanned document images, which illustrates an intermediate point between the bitmap editing and format conversion paradigms" [Bagley and Kopec 1994]. "The central insight behind Image EMACS is that many text editing operations can be implemented directly in terms of geometrical operations on connected components, without explicit knowledge of the symbolic character labels, that is, without character recognition." [Ibid., page 65]. It first analyses the page image to identify connected components, that is, regions of connected "blank ink", which roughly correspond to characters (however, dotted letters map to two connected components and ligatures of multiple letters map to one). Interword space is mapped to a single space character, each line is assumed to be terminated by a carriage return character. Font metrics are estimated. With this minimal interpretation, Image EMACS is able to perform the following operations.

Operation	Description
`cut-region`	Removes characters between cursor and mark; stores text in cut buffer
`paste-cut-text`	Take text from cut buffer and insert into line
`teach-chars`	Lets the user establish a correspondence between connected component shapes and letter names
`key-bindings-from-font`	Binds keys to character images taken from a stored font

`move-to-end-of-line`
`move-to-next-char`
`move-to-next-line`
	Sample cursor movement commands

`Delete-char`
`cut-line`
`cut-region`
`copy-mouse-char`
	Sample insertion and deletion commands

`search-for-string`	Search "by comparing a sequence of reference character images (the search string) to successive character images in the buffer until the mach score exceeds a preset threshold". The search string can be set by typing or selecting characters with the mouse.
`fill-line`	Adjust interword spaces to reestablish right-justification

With its assumption that "all text occurs in a single column of horizontally oriented lines, without any embedded multiline figures, underlining, or vertical rules", Image EMACS can reformat text after editing commands. Among the advantages of Image EMACS, the authors point out, are that what is not edited is preserved (as opposed to OCR software, where what is not recognized is discarded), with no or slight adjustment other languages and graphical notations can be edited, and that image manipulation maps well to FAXes and image libraries.

Because XDOC supplies only word-size granularity information to Multivalent, several of Image Emacs' capabilities are unrealizable ar present. In particular, insertion of new characters is not possible because the physical insertion point cannot be established and the precise rendering of the font of the surrounding text is not available. Given character-level granularity information, it would be straightforward to implement this editing functionality in the more general Multivalent framework. Insertion points could be established with no additional coding. Undo and Save protocols would be made aware that the document can be edited, in the same way as HTML and other formats. Presently reformatting does not reflow text; it makes lines taller and would, if the content were edited as above, make them longer. It would be straightforward to use a linebreaking algorithm, such as used by HTML, to scanned images, with flow breaking across lines, and from text region to region. However, this places greater reliance on correct page analysis and leads to awkward situations when flows cross page boundaries or exhaust text regions. Intrinsically, scanned page images are not the best format for large scale document composition; even the authors of Image Emacs presented it as a means to make relatively small changes or corrections, perhaps when the original document format was lost.

BAMBI

Produced by a European project centered in Pisa, BAMBI ("Better Access to Manuscripts and Browsing of Images") [Bozzi and Calabretto 1997] supports historians and philologists working with digitized manuscripts. Its main display juxtaposes, left and right, the scanned image and its transcription. Since handwritten text is beyond the domain of current OCR software, an editor is provided for the scholar to type in the transcription by hand. Transcriptions are indexed and can be searched, with search hits organized by page. Word spans can be annotated in a subwindow in the main display. Hyperlinks can be added. The transcription is aligned with the image, and clicking on a word in the image highlights the corresponding word in the transcription and vice-versa. The system computationally extracts line and word bounds with a histogram analysis of black and white pixels in horizontal and vertical lines. Data is saved persistently in the SGML extension HyTime format (Hypermedia/Time-based Structured Language) [Newcomb *et alia* 1991], which enhanced the likelihood of interoperability with other systems.

In contrast, Multivalent relies on OCR software (which operates poorly on handwritten text), whereas BAMBI identifies line and word bounds and relies on the user to type in the transcription. BAMBI prefers multiple windows to display transcription and annotation, whereas Multivalent presents associated information in situ, perhaps with a lens to make available alternative views. Now that a later version of Java allows direct access to image pixels, it would be straightforward to write a behavior to identify lines and words by the same method as BAMBI. Another behavior could support transcription, perhaps displayed with existing Multivalent display styles, as transcription hidden behind the image but supporting search or as interleaved between lines; of course transcription could be displayed in a window pane as well.

Text-Image Maps

MIT's text-image maps [Hylton 1994] bring XDOC format OCR and GIF page images to the web. Coming before Java, its interface and implementation are awkward, with HTML image maps and cgi-bin scripts. A mouse click from the user invokes two location-specific actions: treating the corresponding line of text as a citation and looking it up in a bibliographic database, or treating the number at that (x,y) position a page reference and jumping to that page. With more support from the web browser than a simple mouse click, the author of text-image maps foresaw such additional capabilities as looking up highlighted words in a dictionary, showing search hits, adding hyperlinks to index entries, and adding hyperlinks to ad hoc spans selected by the user. Text-image maps, as realized and foreseen, have been entirely superceded in the Multivalent framework.

6.2 Extensible HTML

6.2.1 Problem

As the document markup language of the World Wide Web, the Hypertext Markup Language (HTML) [Raggett *et alia* 1998] is used for hundreds of millions of documents, called pages, and enjoys the support of many tools, tools built for HTML specifically and extensions to long-popular existing applications such as Microsoft Word. Since 1992, HTML has matured to incorporated animated images, forms, tables, frames, style sheets, Java applets, plug-ins, a scripting language (JavaScript), and a document model.

Though meant as a semantic markup language, early deficiencies of HTML led to markup being used to achieve presentation effects. Style sheets have remedied that problem in theory, but the tag set of HTML is too limited to capture the structure of all but the simplest document: across documents its H1/H2/… structures do not map consistently to document chapters/sections/…. The Standardized General Markup Language (SGML) supports tag sets tailored to capture the semantics of individual documents, and the Extensible Markup Language (XML) is a computationally tractable version of SGML. Style sheets describe the appearance of an XML document's particular markup.

Although it is hoped that XML eventually replaces HTML, it will not do so immediately: semantic markup of documents requires time-consuming intellectual work. Moreover, whereas the fixed set of HTML tags makes them amenable to a hardcoded transformation to physical appearance, XML requires an associated translation schema, which is more intellectual work. The most tractable to implement, XSL, lacks flow objects (such as paragraphs with their linebreaking) and presently schema instances rely on HTML flow objects. A more complete presentation language, DSSSL, is very complicated to

implement and has not achieve widespread popularity. Moreover, XML and DSSSL do not address document interactivity. If JavaScript were to be integrated with XML, this would partially address the situation, but JavaScript cannot extend existing capabilities so much as manipulate them and so would be unsuitable for many experiments. Among other deficiencies (see Section 7.3), JavaScript functions do not compose, unless carefully written to cooperate within a limited domain. And beyond that, JavaScript is written by page authors to push effects to readers, but readers cannot easily bring JavaScript to arbitrary pages for read-initiated manipulation.

Aside from its inability to closely describe document structure, the problem with HTML is that it has grown so complicated that it now difficult to experiment with HTML documents or customize them in significant ways. Early versions of HTML were simple enough to be widely implemented, on top of which could often be found a number of new features; though nonstandard and hence requiring that particular browser to view them, the purpose was to experiment with link types, annotation, document visualization, scriptability, or any number of other areas. Now, the effort required to build an HTML engine capable of viewing most web pages has effectively reduced the number of browsers in widespread use to two. With the goal of enabling truly new capabilities, we argue that HTML as designed is not extensible in necessary ways.

6.2.2 Solution

As with support for scanned page images, support for HTML has been implemented without special accommodation in the framework core. The screen dump below illustrating various HTML tags augments others in the Introduction chapter. HTML images is representative of a class of document formats including SGML, XML, RTF, Texinfo, and FrameMaker MIF, in which a formatting engine interprets the content to arrive at a physical layout.

File Edit Style Lens Tool Select Anno CopyEd View Meta Help

<=

Aw, simple **tables** aren't so hard. Here's one with an image and even a nested table (nested tables just worked given the general nesting structure).

0,0	0,1	0,2
more text	more text	
more text with a BR	more text	
0,0 0,1 1,0 1,1		X z
n,0	n,1	n,2

B4

14 El alma no... Alguna vez... Fuera de serie Lente loco

text immediately before

this is an H1

text between

this is an H2

text immediately after

Tom's new ADDRESS

France

a Q element is like BLOCKQUOTE except that language-specific quotes are added

ABBR: some text WWW more text A Sheriff can employ 35 deputies.

EM text **STRONG text** ~~STRIKE text~~ S text **B text** *I text* TT `text` *CITE text VAR text* KBD `text` CODE `text` SAMP `text`

```
some PREformatted text
have to prese
rve line endings,
which usual parsing throws away
Are tags processed with in PRE? no.
```

Does CENTER center every line it encloses, or does it format the block and then center the block? Netscape centers every line individually, which, conveniently, is how justification usually works, as for right jusify.

This is a blockquote. How much is it indented from the surrounding paragraph? Some more text to force more linebreaks so can

1. Ordered list within blockquote: First element
2. Second element. Make this element--time out for a BR--

   longer so can see if left margin is indented by Netscape or not. That should be long enough, except we broke the line and started right back at the left margin. Answer: no.

   - Nested unordered list: First element
   - Second element
3. Back to ordered Third

see margin and not be fooled by a short line. This should be enough but type some more just to be sure. There, four full lines in emacs.

And after BLOCKQUOTE back to normal left margin.

Java Applet Window

The benefits of a Multivalent implementation to HTML (or XML, whose more detailed structure makes available more interesting manipulation) are numerous. Since behaviors can be associated with individual tags, software designers are free to experiment with new tags. They implement the effect of the tag with a (usually) small amount of Java, which is dynamically loaded with the document and therefore immediately available to readers. Since behaviors operate on an abstract document tree, HTML immediately benefits from any behaviors already written for other document formats. Not only can authors associate behaviors with tags and documents, readers as well can do this, for instance, associating a table sorting behavior with all tables.

6.2.3 Implementation: Specializing the Multivalent Protocols

As with the discussion of the implementation of scanned page images, this section describes only how protocols for the HTML media adaptor differ from the general description given in Section 2.3. The HTML media adaptor constructs the document tree in Restore and Build protocols, then relinquishes control to those tree nodes for the remaining protocols.

Restore

Restore parses the HTML document as if it were a linearized tree written in XML, and caches the results. (Documents without corresponding hubs are examined heuristically by the core infrastructure and given a dynamically generated hub, as described in Section 4.4.1, "Cascading and Spontaneous Hubs". This is especially important for HTML, as few HTML pages now have corresponding hubs.)

The typical HTML document on the web is filled with errors vis-à-vis the HTML DTD grammar—improperly nested tags, illegal attributes, a mixture of HTML versions, missing components such as HEAD—and almost all HTML pages take advantage of the ability to legally omit tags implied by the DTD, such as closing paragraph with </P> or ending a list element with </LI>. Both complicate parsing. Of the numerous HTML browsers, almost all have heuristic rules to deal with syntax errors, hardcode the DTD, and, apart from complex structures such as tables, are concerned with structure only to the extent that they can choose the right fonts and linebreaks.

The Multivalent model relies on structurally valid documents, thus necessitating a parser that "corrects" the raw HTML into a structure that is valid according to the DTD. Correcting arbitrary HTML in the media adaptor is a direction for future work; the current system uses the SGML parser SP [Clark 1998] on a proxy server to add implied tags and fix near-correct HTML to produce HTML that can be parsed with a simple XML parser.

Build

Build traverses the cached parse tree and builds the document tree. Some HTML tags are structural (H1, TABLE, FORM), resulting in internal tree nodes, while the others (TT, B, STRIKE) result in spans, and a few (MAP) which provide data for other tags but do not contribute to the human-consumable content, appear as global attributes. Thus the document tree differs from the parse at nonstructural nodes. (The XML style sheet language XSL explicitly distinguishes structural and span tags.)

The aspects of tag formatting that affect the graphics context, such as font, color, and margins, are specified in the HTML style sheet, which are respected by Format and Paint. The effect of some tags can be entirely described by the style sheet. Tags with more complex effects, such as lists, forms, tables, and image maps, are handled by specialized tree nodes.

Format

Format operates as described in the standard Format protocol (see Section 2.3.3), with specialized tree nodes for various tags mentioned in Build implementing tags, such as tables and lists, that differ from horizontal or vertical stacking or paragraph linebreaking. Image nodes take advantage of built-in system incremental reformatting to load images on demand, resulting in faster loads and lower resource consumption when the image is never seen. The limitations of some browsers, including Netscape Navigator, apparently, led to the inclusion in the HTML standard of WIDTH and HEIGHT attributes for images so that formatting could be done in one pass without waiting for image to load. In fact, in the absence of WIDTH and HEIGHT, some browsers stall until the images are loaded, but the Multivalent implementation employs incremental algorithms that can format with a placeholding guess, and efficiently adjust layout when the information does become available.

Paint

Most HTML Painting does not specialize the general protocol. HTML text is painted according to the style sheet and other spans and painted like ASCII. HTML images are encapsulated media (in leaf nodes) that paint their contents as images.

The number of an enumerated list element, however, is not stored in the tree. Since the document tree is strictly structural, the number can be computed and painted on demand by determining the list element node's number within the list parent node. Thus, when list elements are added or deleted during editing, "renumbering" requires only a simple repaint of the list.

Events

Image maps are images with a number of hyperlinks described by geometric shapes, including rectangles, circles, and polygons. The specialized image map tree leaf consumes `MOUSE_MOVE` events and iterates through its list of active shapes asking each, in

object-oriented style, whether the mouse is inside that shape, and if so, the leaf displays the destination of that shape's corresponding link. Image map link hyperlink invocations are similar to span-based hyperlinks: On `MOUSE_DOWN`, the leaf grabs the event stream (see Section 4.5.2, "Grabs") and determines the corresponding shape, if any; if on `MOUSE_UP`, the grab is released and if the mouse points in the same shape, the link is followed, otherwise no action is taken.

Clipboard

The textual version of an image is given as its ALT(ernate) attribute.

Undo

No specialization is made for HTML.

Save

Currently, the URL of the displayed HTML page is written to the hub document. Since the HTML document can be edited within the browser, in the future, HTML could traverse the document tree and generate the HTML markup corresponding to the current document state. Since there will usually be behaviors that cannot, at least directly, be represented by HTML, such as copy editor annotation, HTML Save would either ignore these or, for more important tags, generate facsimiles, which might produce a similar appearance perhaps without the interactivity or robustness to document reformatting.

6.2.4 Related Work

Prior to Java and DHTML, few browsers supported interesting document extension of any sort.

Now HTML documents are customizable to some degree. With Java applets, one can complete control over rectangular portions of the screen. However, Java applets are only coarsely integrated with the document page, as islands of high programmability within the overall page. Since they do not interact with the rest of page, aside from possibly other applets, they are unsuitable for effects that should finely integrate with the rest of the page, such as new hyperlink types or mathematical equation renderers, which would not mesh with the prevailing font and linebreaking and other formatting. Copy editor notes would be unable to open up space between lines in which to write their messages. And applets are unsuitable for effects that combine multiple extensions, such as a highlighted passage with a copy editor annotation. Finally, confined to their rectangle of influence, applets are incapable of implementing lenses. Recently Netscape has released the source code of its Navigator. While free of the restrictions of applets, it is impractical to share experiments and there is no framework to compose extensions (see Section 7.6).

Nor does the combination of the Dynamic HTML (DHTML) document model and JavaScript scripting language achieve sufficient power. While superficially similar to the Multivalent model's behaviors and document tree, JavaScript merely manipulates existing features. While the set of existing features is large enough to accomplish a wide variety of useful tasks, we maintain that to the degree that DHTML departs from the Fundamental Document Model, it limits the potential of experimentation. For instance, it is not possible to implement lenses or executable copy editor markup in DHTML, as these have not been built into HTML; both are implemented as an ordinary extension in the Multivalent model.

6.3 Distributed Annotations In Situ

6.3.1 Problem

Levy and Marshall eloquently state the case for annotation, particularly in situ annotation, as observed in their ethnographic study of information analysts [Levy and Marshall 1995]:

> Annotation is a key means by which analysts record their interpretations of a particular document; in fact, annotation often acts as the mediating process between reading and writing. Analysts generally do not take notes by writing their observations down on a separate sheet of paper or in a text editor .... Instead, they mark on the documents themselves. ... Post-its ... highlight segments of text ... marginalia ... automatically marked text .... These marking practices increase the value of the documents to the analysts and form the basis for their personal and shared files. ... [P]aper is a valuable medium for recording many types of annotations not readily recorded in a digital medium.

Having annotations in digital form would immediate confer upon them the many benefits unadorned documents already enjoy from digitalization. In addition, the digital format would provide the possibility of entirely new forms of annotation, as it allows for the possibility of dynamic annotations, in addition to more conventional passive commentary.

If digital annotations hold great promise, several practical matters must be addressed before they become a fundamental part of the work environment. We suggest that for digital annotations to succeed, they must possess the following properties:

Appearance in situ. Annotations should appear in situ, that is, on the documents themselves (Merriam-Webster's Collegiate Dictionary, Tenth Edition: "in the natural or original position or place"). This property contrasts with the use of newsgroups or email messages, in which portions of documents must be excerpted in order to be commented upon. We interpret this requirement as meaning that the annotation should not refer to or

be part of a copy of a document, which can change independently of the original, but should effectively annotate the document itself.

Highly expressive. Annotation must have the power to engage with the document deeply, potentially modifying content, appearance or runtime properties. This means that annotations must be available at various grains, from the equivalent of a Post-it note to a copyeditor's detailed corrections. In addition, annotations must be able to provide arbitrarily powerful forms of user interaction, providing active capabilities in addition to passive markings.

Format independent. Annotation systems must work with a variety of source document formats. Users should be able to continue to use their preferred document preparation systems, yet produce documents amenable to annotation.

Extensible, yet composable. Individuals, readers, groups, or third parties should be able to develop their own styles of annotations, which may be highly varied [Marshall 1997]. As new annotation types (that is, new technologies) are devised, the new should be seamlessly integrable with the old. That is, not only should previous forms of annotations continue to function as new forms are added, but the various forms need to compose together where appropriate.

Distributed and open. Anyone should be able to annotate any document they can view. Hence, annotating must require no special privileges. Thus, while annotations need to appear as if part of a given document, it must be possible for individuals to create them without modifying the document per se. It must be possible for the annotator to store the annotation wherever that individual has storage capabilities, and make these available by wherever mechanism that individual makes other resources available. No customization of the original document's server should be required for a given individual to be able to make an annotation available. Annotation must be a completely open process, with the resulting annotated document an object existing as a set of distributed network services.

Platform independent. Since annotated documents need to be distributed networked entities, it is desirable to be able to view the document on a machines architectures other than the one on which either the document or the annotation was created.

Robust. As an annotation may reside one place, but refer to a document in another, documents and annotations may change asynchronously. Annotations, therefore, need to be robust enough to survive at least modest document modification.

6.3.2 Solution

A decade ago, Halasz's classic "Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems" [Halasz 1988] identifies "Extensibility and Tailorability" and "Support for Collaborative Work" with annotations as key challenges. The Multivalent Document model addresses the extensibility concern and provides a base

for work on collaboration through annotations. As experimental evidence indicates that users want "to regard annotations as a separate layer of the document ... perceptually distinct from the underlying text" [O'Hara and Sellen 1997], the Multivalent model is naturally suited to annotations where groups of annotations, as by a single author, are collected together as layers that can, as a central aspect of the system, compose with a "base" document and other annotations.

Multivalent annotations meet many of the requirements stipulated above for digital annotations simply by virtue of operating within the Multivalent Document model. Multivalent annotations, which have been described and illustrated in the Introduction chapter, are implemented as or by specialized behaviors. As such, they are composable (the Multivalent framework manages them through the protocols), source format independent (they manipulate the abstract document tree and communicate to encapsulated media types through media adapters), server independent, extensible, seamlessly integrable, immediately portable over the network, and powerful (they have access to every state of the fundamental document life cycle).

Another desideratum of digital annotations is that they appear in situ. We accomplish this requirement by having the individual annotation behaviors rely upon the geometric placement information of document components available in the format stage of the document life cycle. Annotations are attached to a particular component or series of components, and then placed in relation to them. Thus when a document is double-spaced, say, or a table sorted, the annotation is drawn at the right place because it is drawn in relation to the new position. All this is managed by the Multivalent framework calling the annotation at the right time.

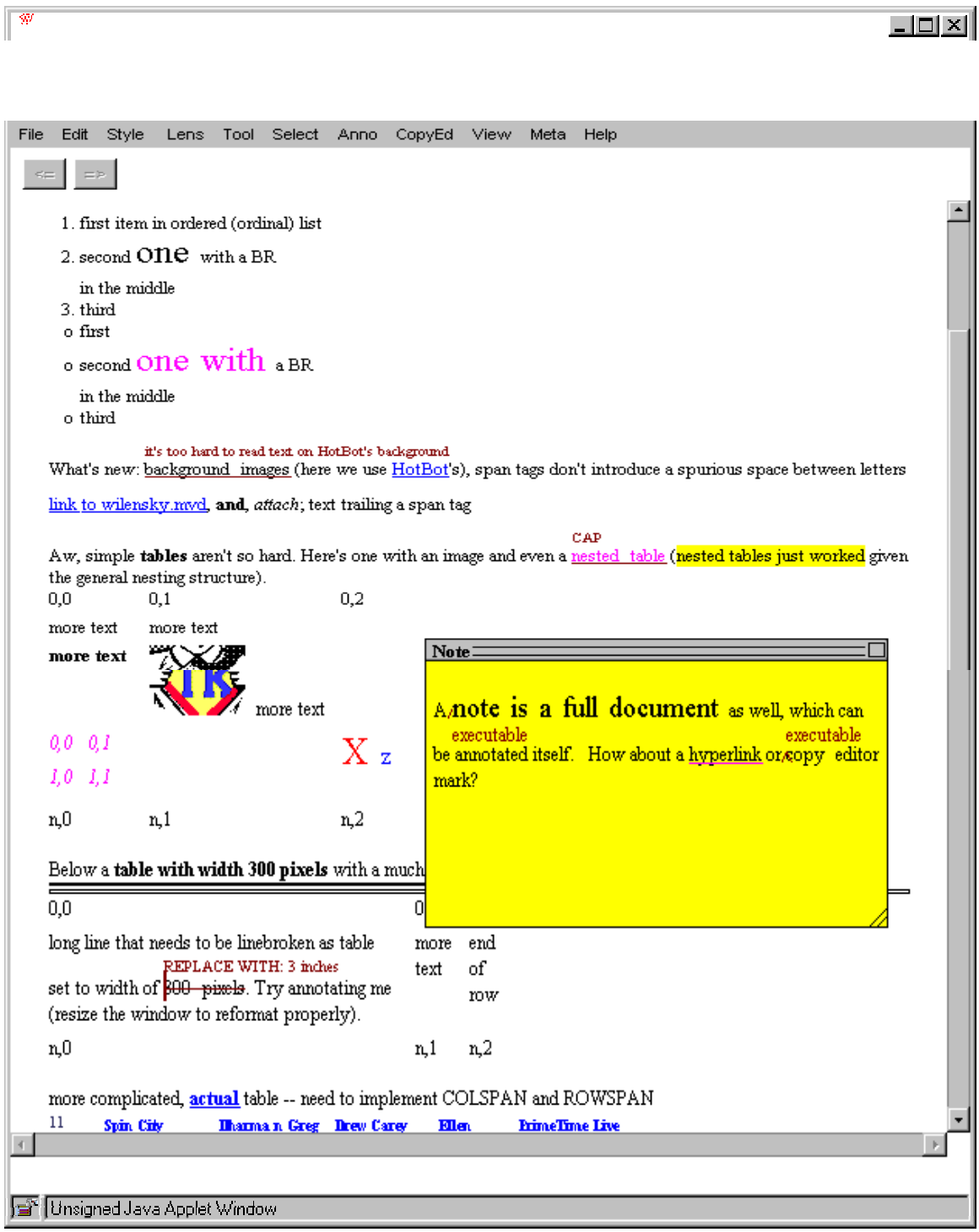
Finally, we require that annotations be robustly positioned. This requirement is addressed via a core feature (see Section 5.2, "Robust Locations"), though it could in fact have been implemented as a shared behavior. Specifically, a standard system class is provided that takes a document structure tree position and creates a redundant description of that place, including its position in the structural tree and offset into the leaf node, an excerpt of the underlying text, a unique identifier of any anchor points, and other information as available. If the document is restored at a later time with the base document or other layers upon which it depends edited, a series of incrementally permissive back-off strategies by the core feature tries to reattach the annotation to the new appropriate location. For instance, if a block of text were deleted before the annotation, the structural tree position may be invalid, but the text will be searched for and, if the excerpted text is unique in the document, the annotation will be placed at the match. If the corresponding text were edited as well, we search for smaller and smaller portions of the text down to some minimum length until a match is possible; closer matches being preferred to those farther away when several matches produce a choice. If every attempt at reattachment fails, as when the corresponding area of the document is deleted entirely, this fact is reported to the user, who can reattach the annotation manually or discard it.

To see Multivalent annotations in operation, the illustrative screen dump from the Introduction is reproduced below. After the intervening explanation of the Multivalent

architecture, it is clear that the annotation types are examples that for one reason or another happen to have been implemented, and that in fact the underlying mechanism supported an open-ended variety of annotation types, each of which can be added seamlessly to the system and distributed as soon as it is implemented.

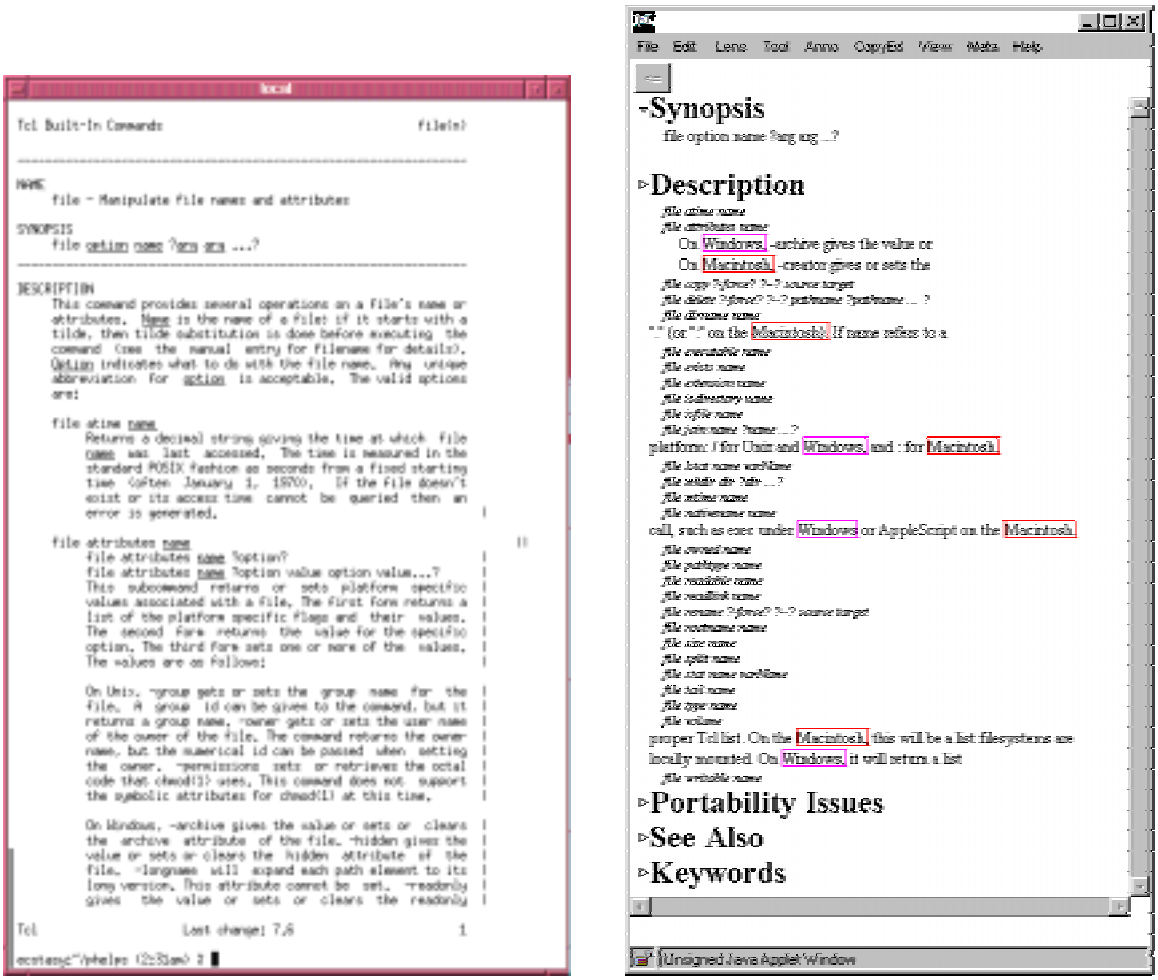
The screen dump below is an annotated version of the one used to demonstrate HTML support above. The Introduction shows an annotated scanned page image, which uses the exact same implementation of the behaviors; here an HTML example is considered in detail. Just under the enumerated list, on the line beginning "What's new" a short textual note behavior has affected the Format protocol to open up vertical space above the line, which in Paint it writes "it's too hard to read text on HotBot's Background". Down a couple lines the text "nested table" is marked with a CAP(italize) copy editor mark. This mark, which has an unambiguous mechanically predictable effect, is in fact executable: when the author receives this annotated copy of this document, clicking on this will capitalize the affected text, remove annotation, and (incrementally) reformat and redisplay the page. Just afterward, the text "nested tables just worked" has been highlighted, as if by a student reading a textbook. Down further on the page, within a table, the text "300 pixels" is struck out and a replacement "3 inches" is suggested. Again, the effect of this annotation is predictable, and the user can click on it to execute it.

Finally, a Note is another annotation type. Whereas the other annotations described here are implemented as spans—point-to-point ranges of text—Notes are implemented as lenses, which are movable, resizable subwindows. Notes have full-featured document trees, and so can have attributed text (such as the larger-sized "note is a full document") and even embedded images (not shown). Notes themselves can be annotated, as shown here with two executable copy editor marks and an intra-document hyperlink (on the word "hyperlink") that scrolls to a point further down on the page.


File Edit Style Lens Tool Select Anno CopyEd View Meta Help
1. first item in ordered (ordinal) list
2. second one with a BR
in the middle
3. third
o first
o second one with a BR
in the middle
o third
it's too hard to read text on HotBot's background
What's new: background images (here we use HotBot's), span tags don't introduce a spurious space between letters
link to wilensky.mvd, and, attach; text trailing a span tag
CAP
Aw, simple tables aren't so hard. Here's one with an image and even a nested table (nested tables just worked given the general nesting structure).
0,0 0,1 0,2
more text more text
more text
more text
0,0 0,1
1,0 1,1
X z
n,0 n,1 n,2
Note
A note is a full document as well, which can executable executable be annotated itself. How about a hyperlink or copy editor mark?
Below a table with width 300 pixels with a much
0,0
long line that needs to be linebroken as table set to width of 300 pixels. Try annotating me (resize the window to reformat properly).
REPLACE WITH: 3 inches
more text
end of row
n,0 n,1 n,2
more complicated, actual table -- need to implement COLSPAN and ROWSPAN
11 Spin City Dharma n Greg Drew Carey Ellen PrimeTime Live
Unsigned Java Applet Window

6.3.3 Novel Annotation Type: Notemarks

Notemarks, a fusion of “note” and “bookmark”, is a form of annotation that provides some of the same ability to sense the overall structure of a document by combining several kinds of annotations together. Prototyped using the Tk toolkit’s text widget [Ousterhout 94], Notemarks applied to UNIX manual pages in the Multivalent framework is shown below, at left a standard manual page display, at right one enlivened by Multivalent.



In the Multivalent interpretation, first structural annotations are used to allow a user to collapse or expanded a section by clicking the section header. However, within otherwise collapsed outline sections, single lines can be visible. These lines are controlled by span annotations of higher priority than the structural annotations, and thus can override the visibility state. Some such spans may be created just for the purpose of overriding a structural collapse. For example, often one refers to a manual page just to check the letter of a command line option, so it is reasonable to pre-configure these lines as visible within collapsed sections. Similarly, the first line of each paragraph of commonly important sections can be excerpted in order to present a highly informative single screen overview. In addition, lines in which the individual user may have

somehow indicated an interest might also override collapse. Note that, in the case of subcommands, command line options and excerpts, the system automatically located the text and applied the span annotation. The highlights, applied by the user, and the search hits, found by the system at user's request, also override the collapsed view of the section. In the screen dump above displays subcommands and search hits as Notemarks.

A Notemark is note-like, in that the collapsed-but-overridden format may comprise a useful summary, and it is like a bookmark, in that clicking on a Notemark causes it to act as an implicit hyperlink, automatically expanding and scrolling to the corresponding section. In short, Notemarks provide a customized display of the overall structure of the document, as if one folded a paper document to display desired portions. Note that Notemarks is not a specialized capability of the Multivalent framework; it is simply a combination of several different kinds of annotation.

The Notemarks idea owes its conception to a fertile environment. To an extensible document system and more precisely to a graphics context controlled by prioritized behaviors, a visibility property was added to implement outlining and speaker's notes. For outlining, a collapsed point hides those lower in the hierarchy. Speaker's notes are generally non-overlapping spans. Exploring the space of combination in this environment that enables easy composition, it was wondered whether it would be useful to allow spans to override the visibility of the hierarchy. The environment also supplied several document types on which to propose this experimental idea. As a result, a new display method was conceived with diverse appeal, some uses of which apply to all document types, others of which exploit specific document genres. It is hoped that the Multivalent environment will increasingly supply interesting features whose mutual friction will inspire new technology.

6.3.4 Implementation: Specializing the Multivalent Protocols

As with the implementation of scanned page images and HTML sections, this section describes only how protocols for annotations differ from the general description given in Section 2.3. In fact, most behaviors can be considered annotative insofar as any behavior can be associated to any document via a hub document, and perhaps further aligned within the document with the Location class. This section considers those annotation types generally considered annotative, such as highlights, hyperlinks, floating notes, Notemarks, and executable copy editor marks.

Restore

Most annotations presently implemented are specialized Spans (see Section 3.1.3). In fact, of the types listed above, only the Note, which is a type of lens, is not.

During Restore, annotative behaviors are instantiated and called to load any associated layer data. Because the associated data for many annotation types is quite small, it is usually placed within the hub document itself.

Since a Note holds a complete document tree of its own, it may be more practical to store that document independently and keep a reference to it in the current hub. In other respects, Notes are like other types of lenses or, internally, like general documents, and thus their operation will not be redundantly described here.

Build

Span-type annotations are saved with location information (see Section 5.2) that is used to robustly reposition them as the document is reconstructed during Build. Since locations were computed on the working document tree during editing, after any tree mutation and layer interweaving that may have taken place during a previous Build, locations are reattached after the mutation and interweaving for the present Build, that is, during the Build After phase.

Format

Span-type annotations that affect formatting do so by modifying the graphics context (see Section 4.3) with a higher priority than the document they annotate. For instance, copy editor markup obtains the space they need to draw their interline comments by artificially increasing the line height. Notemarks forcefully set the visibility property.

Layout algorithms and media types are obligated to respect these requests, as much as possible for a given medium. In this way, annotations are written once, against an abstract document model, as their requests are respected on a case by case basis by all code that affects Format (and Paint). For HTML, respecting a larger line height is accommodated without incident (it is as if the associated character happens to have a larger point size). For scanned page images, for which page appearance is "fixed", the current positions of content below the annotation is shifted down by an amount equal to the increase.

Paint

Much of the change in appearance due to annotations is achieved with the same modification to the graphics context as in Format, changes which, again, media types are obligated to respect. Highlights set the background color to yellow, hyperlinks set text foreground to blue and add an underline, and deletion copy editor marks add an overstrike line. Many copy editor marks further draw text and arbitrary graphical objects such as an insertion caret.

Events

Hyperlinks and copy editor marks are initiated with a mouse click, and in the future all spans with accept a qualified click (say, Control-click) and present a menu to edit their properties, including deletion of the span. The annotations presently implemented do not specialize event handling more than that described for the general Event Protocol (see Section 2.3.5).

Clipboard
No specialization.

Undo
No specialization.

Save
Heterogeneous collections of annotations can be saved together a layer apart from the annotated document. Annotations are saved in a hub document with multiple types location positioning information. The hub document is in XML format, and annotations can save further information (the text of a comment, the destination of a hyperlink) as an XML attribute, XML content, or as a separate file, with only a URL saved in the hub itself.

6.3.5 Related Work

The following table compares the representative annotation systems Acrobat, MarkUp, ComMentor, and Knowledge Weasel to Multivalent along the desirable properties for annotations describe above. It also includes as a reference point OpenDoc, a system with no annotation capability per se but sophisticated extensibility to show what annotation systems add beyond an extensible document model. Scores are given in the range of zero to one, with zero indicating no support, and one excellent support.

Attribute	**Acrobat**	**Mark-Up**	**ComM**	**KW**	**Multi-valent**	**OpenDoc**
In situ	1	1	0.5	0	1	0/1
Highly expressive	0.5	0.5	0	0.2	1	1
Format independent	0	0	0	1/0.1	1	0
Extensible/composable	0.1	0.1	0	0.8	1	1/0
Distributed/open	0.25	0.5	1	0.5	1	0
Platform independent	0.9	0	0.1	0.25	1	0
Robust	0	0	1	0	1--	n/a
Presentation Styles	0	0	0.5	1	0	0
Database	0.1	0	1	1	0/0.25	n/a

Adobe's Acrobat [Adobe] bears a number of similarities to the scanned page image application of the general Multivalent model. Adobe has published the specification of the PDF format [Adobe 1996] viewed by Acrobat, and that format is in principle extensible by anyone. In practice, however, it is extensible only by Adobe as extensions to the format require corresponding changes to the viewer, and that is proprietary to Adobe, and, unlike early HTML viewers, difficult to build. Moreover, the types of

annotations provided in Acrobat, though growing with each new version, are geometrically positioned at x,y coordinates on the page, not tied to content, and therefore not robust to changes in the document.

MarkUp [Mainstay] "images the document and creates a version that looks exactly as the original would, if printed on paper". It is representative of large class of annotation systems that includes Hot Off the Web [Insight Development] and that is not extremely different from Acrobat, that take a picture of the appearance of a document, and provide any number of graphical drawing tools to edit the image. In particular, MarkUp provides a magnifying glass, strikeout, an arrow symbol, a highlighter, text extraction, oval and rectangle shapes, a pop-up note, proofreader's marks, QuickTime video, and voice annotation. It can merge annotation overlays. Because it captures the page appearance by posing as a printer device, it works on all source document formats (though this trick appears to be limited to the Macintosh). However, document content is reduced to a series of graphical shapes and text, and therefore it cannot perform any structure-based manipulation, such as sorting tables. MarkUp is not suitable for long-lived annotations such as academic commentaries as it is not robust to changes in the original document. Finally, since MarkUp annotation is divorced from the original document, revising the original in light of annotations requires side-by-side presentation of both documents, at which time annotations are not in situ.

ComMentor [Roscheisen *et alia* 1995] focuses on the server side of annotation support. ComMentor has a complete and well worked out meta data strategy and a database system for managing annotations, but could only provide minimal functionality at the client as it relied on taking the source code for Mosaic [NCSA] and enhancing it; Mosaic was a moving target and has now been entirely superseded by commercial browsers-whose source code is not available. ComMentor would be a nice complement to Multivalent annotations.

Knowledge Weasel [Lawton and Smith 1993] distinguishes between surface annotation and deep annotation. Like ComMentor it focused on database aspects like a common record format and surface annotations and not as much on deep annotations, except for spatial data imagery. It took advantage of common tools (Tcl and Tk) for wide availability, but ultimately it was also limited to them. If the text widget does not admit, say, lenses, then that is an insurmountable barrier to implementing them. It presents annotations in separate windows, as do Mosaic and Microsoft Word.

ComMentor and Knowledge Weasel recognize that it takes a great deal of effort to build a document formatter-renderer, and hence follow a strategy of interoperating with existing formatter-renders. Unfortunately, annotation requirements push against the limits of such systems. Instead, we pursued a strategy makes it more difficult to take advantage of existing applications, but, we hope, will prove sufficiently general to enable the incorporation continuing innovations in digital document technology.

7 Related Work

Work related to the applications discussed or specific features has been described previously, in the corresponding section. This chapter considers work related to the Multivalent architectural approach, primarily concerning extensibility of digital documents. Related work is considered along several key dimensions, contrasting it with the Multivalent approach, listing one or more systems that share that approach, and considering at length the one system considered the superior technological exemplar of the group.

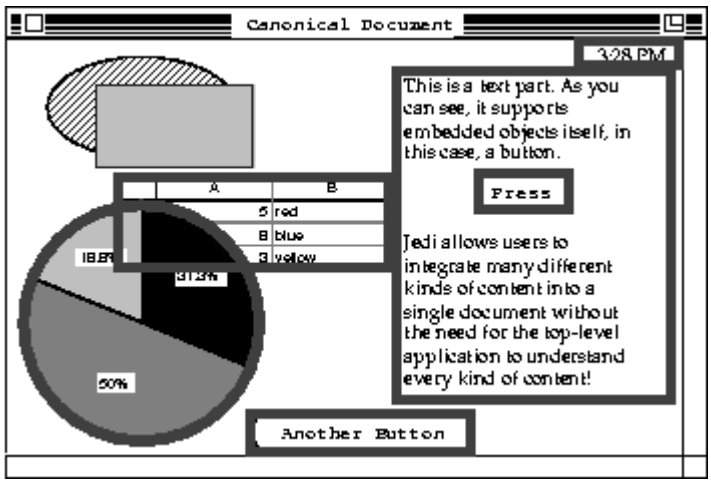
Recall in the following arguments the Multivalent definition of extensibility: *power* to control and extend existing functionality, *flexibility* to accommodate unanticipated experimentation or innovation, and *composition* of extensions to produce a coherent whole at no disadvantage to monolithic systems conceived all of piece.

7.1 Integration vs. Juxtaposition: OpenDoc, OLE, Quill, HTML with Java Applets and Plug-ins

Apple's OpenDoc [Apple 1995], Microsoft's Object Linking and Embedding protocol (OLE; now known as ActiveX) [Brockschmidt 1995], Quill [Chamberlin *et alia* 1988], and HTML extended with Java applets and plug-ins all open the document to arbitrary extensibility through a system of nested editors. Various media types and their associated editors or viewers can be layed out alongside or nested within each other (in the case of applets and plug-ins, nesting is limited to a single level). As an illustration, the OpenDoc

Technical Summary [Apple 1993] describes the user experience of a prototypical OpenDoc application as follows:

> In this example document [below], the root part is an object-style graphic editor. In the upper left corner, there are two content objects, a rectangle and an ellipse. A clock part has been embedded in the top right corner. Towards the center and bottom left, a chart part is embedded in the text part. A second button part is embedded in the graphics part, at the bottom center of the document. In the illustration below, a thick gray border is drawn around each part.



The major advantage of this approach is that by adhering to the protocols of embedded media editors, the programmer can introduce potentially any new data type, including ones for data types conceived after definition of the basic framework. Moreover, in this "document-centric" approach as opposed to application-centric approach where applications "own" their data types, the data type is considered primary so that different editors can be swapped in as better editors for that media type are developed. In this way, this approach is a major advance over the monolithic approach.

The problem with the nested editors architecture is that it does not support extensibility of individual editors themselves. Consider two text editors, one that understands HTML; the other, annotation. Now given HTML content that the user would like to annotate, neither editor suffices. All desired features are available in one editor or the other, but not simultaneously. Unfortunately, editors must be replaced wholesale, the features and limitations of one exchanged for another's similar mixed bag.

The Multivalent approach fosters behaviors that do not duplicate each other, but rather that leverage one another, supplying what is lacking. In the Multivalent model, extensibility is much more fine grained, extending into the media editors themselves.

7.1.1 OpenDoc

OpenDoc, despite its undeserved status as an abandoned technology, is the most sophisticated implementation of this class of nested editor systems. OpenDoc calls its editors *Parts* and defines an architecture for the incremental addition and coordination of new Parts. OpenDoc shares many goals with Multivalent Documents, from the document-centric orientation to a roughly similar set of protocols for extensions, and of all other digital document systems is the one most spiritually compatible.

The following table roughly matches key elements of the OpenDoc infrastructure with their Multivalent equivalents; a comparison of protocols is found in the next section.

Characteristic	**OpenDoc**	**Multivalent**
Document model	Nested editors	Attributed hierarchy
Unit of extensibility	Part (Editor)	Behavior
Modify behavior of existing code via	Can't without source code	Before/After protocols
Modify appearance and behavior of region or range of content	Can't in general (editors may implement individually)	Lenses, Spans
Platform independent?	Yes (mostly; with multiple compilations)	Yes
Language	C++	Java
Lightweight programmability	Scripting	Future work
Versioning	Drafts	Future work
Storage	Structured storage	Files, later structured storage in Zip format
Control flow	Library	Framework

In common with other systems in this group, OpenDoc has a relatively coarse granularity of extension, the editor. Editors may be nested or swapped for another that can manipulate the given data type, but editors themselves are not extensible, and actions such as Multivalent lenses and spans that can cross editor boundaries are not supported or at best are supported internally in the occasional editor.

OpenDoc abstracts away the native computer operating system, for the most part, so that Parts can operate on any OpenDoc platform with a simple recompilation. Specifically, menus, windows, user events, storage, data transfer (clipboard and drag-and-drop) are all platform independent; but drawing primitives are platform dependent. To the Multivalent implementation, Java brings platform independence in all these areas. The Multivalent framework takes advantage of the platform-independent character of Java to dynamically load behaviors, a feat possible as well in OpenDoc, at greater effort, requiring a C++ compilation per platform and platform-specific coding of drawing primitives.

OpenDoc is more mature in several areas not part of the core Multivalent architecture, but nevertheless closely related. OpenDoc takes advantage of Apple's Open Scripting Architecture (OSA) for basic programmability including control flow, and furthermore defines vocabularies for data types (charts, spreadsheets, text, et cetera) so that scripts can speak in terms of high-level entities of that data type. The current Multivalent implementation, it is believed, has the necessary hooks for manipulation by a scripting language, when an existing scripting language is chosen. OpenDoc supports document versioning, an area where Multivalent will interface with existing systems, and OpenDoc uses a sophisticated structured storage system for managing all pieces of a document, a component packaging scheme Multivalent will evolve on top of the popular Zip structured storage format, support for which is built into Java.

7.1.2 OpenDoc Protocols

Like Multivalent, OpenDoc defines a set of protocols to which extensions must conform, many of which are equivalent across the two systems.

OpenDoc Protocol	**Description**	**Multivalent Equivalent**
Layout	Frame and facet manipulation (space negotiation)	Format
Imaging	Drawing Part in each of its frames	Paint
Activation	Includes changing ownership of selection, menu, keystroke focus	Setting grab, as far as applicable
User Events	(similar to Multivalent)	Events
Semantic Events	(same as Multivalent). Used in scripting	Future work
Storage	(similar to Multivalent)	Restore and Save
Binding	Mapping part editor to document parts	Not applicable
Linking	Communication of live, shared data between parts	Could be arranged between individual behaviors with a manager behavior
Embedding	Implemented if part can embed other parts	Built in if media adaptor exposes structure of content in tree
Clipboard	(same as Multivalent)	Clipboard
Drag-and-drop	Including data conversion	Individual behavior can implement Java drag-and-drop
Undo	(same as Multivalent)	Undo
Extensions	Part-specific interface for use by other parts	Not needed; flexibility inherent in Java

The two models are more similar than different. In some areas Multivalent is more radical, while in others OpenDoc is more mature. Layout and events most clearly show the differences within the similarities.

Both models approach layout and painting similarly in order to accommodate arbitrary media types. Media content is modeled as a hierarchy, with Multivalent placing all media at the leaves and OpenDoc allowing media at internal nodes, in editors within which are embedded others. The extension associated media type itself, not the framework, is responsible for painting the media content (as well as editing and storage). Multivalent is more fine grained insofar as media can be interleaved with other media, but OpenDoc mitigates this advantage somewhat with its more sophisticated viewing model. OpenDoc parts have corresponding visual counterparts called *frames*, and a single part may have zero, one, two or more frames embedded in other parts, showing different regions of the same content, or different representations (also called appearances or presentations) of the content.

OpenDoc's dispatcher propagates events to the proper part editors, based on event location and ownership. For nested parts, OpenDoc follows and "inside-out" activation model, giving ownership to the most deeply nested part first, from which the user can expand the range to enclosing parts. In Multivalent, events are not sent directly to a media adaptor but are propagated along the document hierarchy during which they can be filtered or short-circuited. Usually the event will reach a media adaptor, but OpenDoc lacks a mechanism by which event dispatch can be customized, as for instance in adjusting the coordinates of a mouse click inside a magnifying lens.

7.1.3 Quill Protocols

Since the Multivalent model proposes a set of protocols that are expected to remain constant and sufficient for future diversity, it can be instructive to examine an older system's protocols to measure protocol volatility over time.

Developed in the late 1980's, Quill is "organized as a collection of cooperating editors" and shares with Multivalent the "desir[e] for specialized types of material to be manipulated by editors with specialized knowledge; yet … overall … present the various materials to the user in a uniform, well-integrated way" [Chamberlin 1988].

The following table presents the procedures required of a Quill constituent editor.

Quill Procedure	**Description**	**Multivalent Equivalent**
Broken	Content is "broken"; must be reformatted and redisplayed	Done by nodes of document tree
Click; Key	Mouse button event; keystroke	User Events
CreateMaster	Create a nested editor instance	n/a
CurrentLNode;	Become the active editor; no longer	n/a

Reset	the active editor	
Destroy	Delete a subtree from the document tree	Not a protocol; done on shared document tree
Init	Initialization of editor (as separate from content)	part of Restore
Locate	Ask for the physical position of a given node within the document	Done by nodes of document tree
MakePrint	Generate printable output file	Paint/Print
MakeSGML	Generate an SGML file for later editing or interchange.	Save
NewView		System + Paint
Redraw; Repair	Redraw node and descendants; redraw portion of parent node	System repaint + Paint
Work	Background work thread, usually formatting	Future work?

In terms of Multivalent protocols, Quill has a version of Restore (Init), Build (CreateMaster, with the rest internal to the editor), Paint (Redraw, Repair, MakePrint), Events (Click, Key), and Save (MakeSGML). Format is internal to editors. Since editors are placed in a tree structure, it externalizes as procedures functions (Broken, Destroy, Locate) that are handled by more generic Multivalent tree nodes. Overall, general document function has remained consistent in the ten years between Quill and Multivalent, with most change on protocols due to the change from editors to more fine grained behaviors.

7.2 Composition vs. Shared Library: GNU Emacs

An extremely popular text editor, Emacs [Stallman 1981 and Stallman 1997] supports a large number of significant extensions with a comprehensive library of text manipulation functions. Its extensions, some of which happen to be packaged with the base system, include a USENET news reader, multiple e-mail handlers, a reader for the hierarchical Texinfo documentation format, a calendar and appointment manager, a source-code debugging interface, a directory editor, and modes for editing most all major programming languages, among many others by many different authors.

Based on the TECO editor, Emacs has been under development since 1974. This, in part, accounts for its great robustness, ports to a wide variety of platforms, and its large number of extensions. But Emacs also has an historical flavor. Although a menubar has been recently added, Emacs is primarily controlled with the keyboard, with multi-key, semi-mnemonic key bindings invoking commands. Emacs recently incorporated support for non-English languages, but does not yet support proportionally spaced text, much less multimedia formats. Nevertheless, Emacs remains a high standard of extensibility.

The following table roughly matches key elements of the Emacs infrastructure with their Multivalent equivalents.

Characteristic	**Emacs**	**Multivalent**
Unit of extensibility	Major and minor modes	Behavior
Modify behavior of existing code via	Hooks	Before/After protocols
Implementation language	C and Elisp	Java
Extension language	Elisp	Java
Simplest means of customization	Local variable binding lists	Attributes
Document model	Lines	Attributed hierarchy
Modify appearance and behavior of range of text	Text properties	Spans
Command invocation	Keymaps	Behavior via Event protocol

7.2.1 Extensibility

Composition

The key difference between the extensibility of Emacs and the Multivalent model is the focus on composition. In Emacs, a specifically coordinated set of Elisp functions (see below) make up a major mode, which is an application such as a news or mail reader. Exactly one major mode is in effect at a time. Major modes can be enhanced with any number minor modes. Although potentially an unbounded set, minor modes are few (namely: autofill, autosave, fill, font lock, hscroll, ISO accents, outline, overwrite, abbrev, icomplete, line number, resize minibuffer, scrollbar, transient mark).

In other words, the Emacs programmer builds applications using the standard library of text manipulation functions independently of other Emacs applications—major modes do not compose. Minor modes do, but, as the name suggests, their effect is small, and the manual for the extension language Elisp [Lewis *et alia* 1995] warns that "a minor mode is often much more difficult to implement than a major mode. ... Often the biggest problem in implementing a minor mode is finding a way to insert the necessary hook into the rest of Emacs".

In the Multivalent model, behaviors are self-sufficient units at all times meant to compose with others, if not always productively at least never deleteriously. Applications are built out of sets of behaviors. In contrast to major modes, behaviors can be usefully used in different applications. In contrast to minor modes, behaviors can have significant effect, and hook into the rest of the system by design.

Hooks

Both Emacs and the Multivalent model support observers on interesting document activities. Like the Multivalent model, Emacs supports "before" and "after" hooks, where applicable. Emacs hooks are often high level semantic activities, as can be seen in this partial list:

activate-mark-hook
after-change-functions, before-change-functions
after-init-hook, before-init-hook
after-insert-file-functions
after-make-frame-hook, before-make-frame-hook
auto-fill-function
auto-save-hook
blink-paren-function
c-mode-hook
calendar-load-hook
command-history-hook
comment-indent-function

Thus, the Emacs programmer is given an obvious place to attach an extension. If the extension modifies calendar-related functions, `calendar-load-hook` can be quickly identified as the desired hook. On the other hand, this scheme relies on the system to provide all hooks that may be required.

In contrast, Multivalent observers are tied to low-level document protocols, often on document subtrees. The behavior writer must determine that, say, a mouse down event on a the first child of a node named TABLE should initiate a table sort. The combination of protocol and subtree, however, sometimes implies a higher semantics. (In the future, the Multivalent model will be extended to explicitly support high level hooks, implemented like the signal capability of lenses, which behaviors act upon if recognized, harmlessly ignore if not.)

Semantic vs Low-level Extension

Overall, not only with hooks, Emacs aims to explicitly identify and provide support for all interesting activities. For work within its capabilities, this makes programming easier than dealing with low level features. However, Emacs' emphasis on the specific rather than the general means that unanticipated needs often are not accommodated well if at all. In contrast, the Multivalent model opens the fundamental protocols—the lowest level—for greatest potential, and higher level operation could always be provided by code libraries layered on top.

The lack of pervasively general capabilities in Emacs means that certain applications require specific accommodation, which the author may or may not be granted. As an extreme example of this, Emacs explicitly provides "the overlay arrow", which is used by the source level debugger to point to the current source line without modifying the program source text—a feature of limited applicability. This feature has been superceded by the "before" and "after" text of overlays (overlays are like spans that live in a separate

layer than their associated text): text may be show text before and after the overlay—yet again a feature of limited applicability. In the Multivalent model, an overlay arrow is a trivial application of the After phase of Paint protocol.

Key Bindings

Emacs maps user keystrokes to commands via keymaps, which may be nested for multi-keystroke mappings. In the Multivalent model any behavior can accept events. In both systems, finding a unique key binding can be cumbersome. Emacs uses potentially long, potentially only slightly mnemonic key sequences to trace out a unique path, or the user can type in the long but understandably named name of the command. In the Multivalent model, keys can shadow one another without warning.

Programming Language

Emacs is implemented in C for high performance primitives and a dialect of Lisp called Elisp [Lewis *et alia*1995] for higher level control, and is extended in Elisp. Because the Elisp interpreter is built into Emacs, most of the body of Emacs itself and all Emacs extensions port to other platforms with few or no changes. Recall that the Multivalent framework has been implemented in Java, and extensions are implemented in Java, yielding the same benefits. Both Elisp and Java have byte-code interpreters, and the intense commercial interest in Java has led to so-called just-in-time (JIT) compilers as well as native code compilers.

Dynamic loading, the most important characteristic of an extension language, is shared equally by Elisp and Java. Cross-platform execution is helpful for distributing extensions, and this too is shared equally by the two.

Since Elisp can be written and immediately evaluated within Emacs and, as a dialect of Lisp, is written a function at a time and does not have variable type declarations, it has the feel of a scripting language. By contrast, Java is a highly structured compiled language, though very fast Java compilers make for rapid edit-compile-test cycles. Although the present author has little experience writing Elisp and limited experience with Lisp, the use of Elisp or Java to write extensions probably makes little difference to programmers: extensions tend to be of medium size, large enough to benefit from preplanning, discipline, and structure in managing their numerous interrelated pieces, which mitigates Elisp's rapid prototyping scripting feel; but not so large that Java's superior support for large programs becomes significant.

One advantage of a scripting language over a compiled language, Elisp over Java, is that small changes can be made more simply. Elisp places variable settings and function definitions in a global address space. Extensions can easily change variables (characters between tab stops), including key maps (tab key to insert the right amount of spaces rather than a tab character), and override functions. In Java, any change that must be made in code (outside of a file of preference settings) requires more overhead. There is the syntactic overhead of defining a valid Java class and method, and multiple changes

are more difficult to combine together, resulting in a more cumbersome set of small files to manage.

Elisp defines as part of the language that each variable and function be given a documentation string. Documentation strings are collected centrally, where they can be searched, and since Elisp programs generally use only built-in functions and do not extend other extensions, the programmer has a relatively complete set of documentation. Although each documentation string is brief, that combined with a typically long, descriptive name for the variable or function is helpful. Since Emacs itself is almost always used as the programming editor to write Elisp, documentation is conveniently available. Java defines a standard syntax for associating documentation with variables. Extracted documentation is made into web pages, and the programmer relies on the searching and other capabilities of the web browser. More than specific technical means of supplying documentation, Emacs extension writers benefit from the maturity of the system. Informed by experience, documentation can counsel against use for one purpose and recommend another function instead. Since extension writers interact with the central system and less so with other extensions, there is a controllable body of documentation amenable to quality control. The Multivalent model encourages distributed development not amenable to such a degree of control; one could hope to establish standards of documentation that encourages third parties to invest likewise.

Of course, the general user does not program extensions to the editor, but he may customize it. Much customization in Emacs is conducted through variable settings, and the latest version of Emacs (version 20) can set these variables through a series of menus. Similarly, Multivalent behaviors respect numerous, behavior-specific attributes, and there are plans for a Preferences panel that unites contributions from behaviors in the same way as the general Multivalent user interface composes menu needs.

7.2.2 Document Model and Typographic Display

Emacs represents text as lines. Characters are positioned at a row and column intersection. Emacs does not support finer positioning, proportionally spaced text, or geometric shapes, much less multimedia. This leaves the typographic quality of the display at the level of decades-old terminal displays. (On the other hand, Emacs runs well on decades-old terminals.) Such a simple text model is conceptually easy to understand, but leads to awkward handling of more complex structures such as tables or nested structures. Table formatters must model the non-linear, two-dimensional table internally, and then render it as lines, which involves an awkward, line-by-line linearization across table cells in the same row. The document model and typographic capability of Emacs show its age.

Emacs associates display properties and event handlers with ranges of text with text properties, which are very similar Multivalent spans. Text properties include:

Display properties: category (default property values), face (font and color), mouse face (when mouse enters), and invisible;

Formatting properties: hard/soft newline flag, left margin, right margin, and justification;

Event properties: local key map, read only, intangible (point cannot be place there), modification hooks, insert front/behind hooks, point entered/left hooks

Compared with Multivalent spans, Emacs text properties for display and formatting are necessarily more limited, constrained as they are by a more limited document model. Among the event properties, Emacs has identified and given direct support for a fixed number of cases, presumably all the common cases, such as text modification. The Multivalent model gives lower-level access to a wider range of events; this leaves potential for unanticipated event patterns. This leaves the higher-level events of Emacs to be synthesized from a pattern of low-level events. In the future the Multivalent model may incorporate the insert front/behind and modification hooks and other high level events that cannot be reliably captured with low-level events.

7.3 Deep Extension vs. Scripting an API: Dynamic HTML, Microsoft Word, FrameMaker

Dynamic HTML, Microsoft Word, FrameMaker and numerous other digital document systems expose an application programmers' interface (API). Many useful applications can be written within the API (if not the API would be uninteresting), but the document facilities themselves cannot be extended, leaving the issue of how to compose such extensions moot. Compared to Emacs, the document facilities are much more modern, but like Emacs the programmer is ultimately limited by those same built-in document facilities.

7.3.1 Dynamic HTML

Dynamic HTML (DHTML) (defined differently by [Netscape 1998] and [Microsoft 1997]) is a Frankensteinian admixture of HTML, CSS, and JavaScript, with a Document Object Model (DOM). Through the interfaces defined in the DOM, JavaScript can manipulate the logical tree structure of the HTML document and change the document presentation through CSS properties.

HTML is a specific set of tags that does not capture the semantic, logical structure of a document, despite its stated goal of semantic as opposed to physical markup—does an H3 tag mean a chapter, a subsection or, as its often used, a title in a medium-sized bold font? A better alternative would be SGML, which can support tag sets tailored to the document or document class and so, as all introductory writing on SGML states, makes an informative base for manipulation and a durable description for archival storage. XML

retains all the benefits of SGML while dropping its unnecessary complexity and so, at least for text, makes an excellent format for web documents. A principal downside to SGML or XML is that its semantic markup requires additional thought on the part of the author, and a one-time though not insignificant cost of writing a corresponding style sheet, if the document is to be viewed in a custom way.

CSS, described above vis-à-vis Multivalent style sheets (see Section 4.5.3), controls properties of document entities that match a (usually) structural pattern. CSS Level 2 is relatively powerful, controlling not only fonts and spacing, but also floating regions, absolute positioning, layers, some generated content (automatic numbering, for example), and paged media. Unlike DSSSL, Proteus, and other presentation languages considered in the section on the Format Protocol (section 2.3.3), CSS is not capable of defining new flow regions, but its built-in types cover the most common cases and overall CSS2 strikes a creditable balance between complexity and power.

JavaScript, which is not related to Java beyond superficial syntactic similarities, provides elementary types, expressions, control flow, and functional abstraction, and access to entities in the HTML page presented as objects with properties and methods. Although JavaScript was developed for very short scripts tied to pieces of forms and was plagued by fragile implementations, it has developed into a reasonable if unexceptional scripting language for short scripts, though personal experience and a casual survey of other web authors suggests that JavaScript's widespread use is due exclusively to the fact that it is the only means of achieving various runtime effects in HTML, rather than any quality as a language per se. Recently both Netscape and Microsoft have proposed factoring out JavaScript from necessary incorporation in the page in which it is used, into separate files that can be shared by pages. In this case, JavaScript looses one point of distinction with Java, and more applications would more logically use the superior language Java, a language whose design is widely praised.

The Document Object Model is "a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The DOM provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them" [Wood *et alia* 1998]. DOM Core Level 1 defines the object types Document, Node, Attribute; and node subtypes Text, Comment. HTML Level 1 "exposes a number of convenience methods and properties that are consistent with the existing models and are more appropriate to script writers"; basically for each attribute in the HTML DTD, DOM defines a field in a corresponding object, to which language implementations provide get and/or set as appropriate.

Presumably the technologies HTML, JavaScript and CSS were thrown together as the pillars of DHTML due to their ready availability in the most popular web browsers. Nevertheless, HTML and JavaScript are weak representatives of their respective classes, CSS is good if limited, and DOM is considerably less flexible than the Multivalent

document tree. To propose the witches' brew of the four technologies as a new standard for future dynamic manipulation of web pages in the client is folly.

7.3.2 Microsoft Word, FrameMaker

FrameMaker and Microsoft Word, as representatives of word processors with APIs, have each been under development for more than a decade, and by now one could presume that both do most of the things most people need to do most of the time. Each has many hundreds if not thousands of functions, only a fraction of which the ordinary user uses or even knows about, and yet on the other hand, specialized needs that appeal only to a small audience are unlikely ever to be supported. Call it feast *and* famine.

Both FrameMaker and Microsoft Word do expose APIs, but, as with Emacs and Elisp, one cannot extend the functionality so much as orchestrate existing functionality; that is, the problem is that the system can be customized only to the extent previously anticipated. In most places, the APIs allow external control at a relatively high level ("insert video at given position"), which is good for high productivity use of the system. Unfortunately, external control is limited to manipulating existing features, not extension of the features in novel or experimental ways unanticipated by the system designers: There is no "lowest-level" interface to the Lifecycle (no access to paint with which to implement lenses, no clipboard for pasting alternative bibliographic representations). Second, even if "lowest-level" access is available (often events are available), given a number of customizations, current systems do not allow them to be active simultaneously, as they compete for the same system resources and hence conflict with each other, with no means to resolve these conflicts.

7.4 Client vs. Server

In a client-server system, jobs that involve reporting from large data sets located at the server should be done at the server, and jobs involving interactivity with the user should be done at the client. For instance, most document annotation involves intimate interaction between user and system and so belongs in the client. Present web systems lean too heavily toward the server, implementing something as common as an outliner display there, because the server is easily programmed with scripts, while the browser is not. Historically JavaScript could not modify the page once it was constructed; the introduction of DHTML (see above) should alleviate this limitation, though DHTML can only transform HTML into other HTML and therefore is not fundamentally different, just faster.

The Multivalent model is biased toward intelligence in the client rather than the server. Although the server can deliver specialized behaviors along with its documents, in the main the user manages which behaviors are active for which document genres. We claim

that this is the most efficient means of improving the cooperative client-server system. For a user to add new functionality, say versioning of all web pages seen, a new behavior is added to the client. To add this same functionality on the server side would require the cooperation of the hundreds of thousands of web servers in the world. Moreover, to take advantage of multiple document filters, it is more efficient and practical to coordinate them at one place, the client, as Multivalent has done, rather than trying to link multiple servers.

7.5 Union vs. Confederation, or Build vs Reuse: Microcosm, UNIX Guide, Firefly

Every computer media type comes in numerous data formats. Images can be of format GIF, JPEG, Macpaint, FlashPix, pnm, BMP, PNG, Embedded PostScript (EPS), or TIFF, which itself has numerous subtypes, and so on. Videos can be MPEG, avi, QuickTime, Real Video, Net Show, NEXTIME, and so on. Audio can be au, wav, MP3, and so on. Documents can be ASCII, troff, LaTeX, FrameMaker, RTF, HTML, SGML, PDF, and so on. Writing a viewer for most data formats is nontrivial, and so many research projects chiefly concerned with other issues, such as hypermedia and media synchronization, have relied on existing viewers. The developers of UNIX Guide [Brown 1992] state the problem as follows:

> To survive, any hypertext system needs to interact with and to use other tools. A choice facing the designer is which of these alternatives to select:
> (a) to use what is there already, i.e. to use the existing command languages, editors for various media, spelling checkers, printing programs, searching tools, etc.
> (b) to provide a framework by which others can plug extra tools, specially written to certain conventions, into the hypertext system.
> …
> Clearly approach (b) is more attractive for a researcher keen to escape from the dreary current world into a brave new world. However it requires writing enough supporting tools to make the system viable. Then, with luck, the system will take off, and users all over the world will add supporting tools to it.
> For a small project, the developer or researcher needs to grit his teeth and adopt approach (a).

For example, Microcosm [Fountain *et alia* 1990 and Davis *et alia* 1992] builds a hypertext system on top of existing applications, including Microsoft Word (before it had built-in hyperlinks). Firefly [Buchanan and Zellweger 1992] concentrates on satisfying temporal synchronization constraints in multimedia content, combined with asynchronous events such as user interaction. For low-level media display, it relies on existing media players, imposing the minimal requirements that they can be controlled to start, stop, pause and resume playing.

The philosophy is that, rather than compete with already available, heavyweight applications such as Microsoft Word and Excel, one should take advantage of the hooks

these applications make available and layer the new system on top of them. While gaining the power of high-feature applications, this strategy is limited to the extent that such hooks are permitted by those applications. If one hopes to accommodate innovative new technology, it is unlikely that these hooks will prove sufficiently versatile indefinitely. In practice only the most basic demands can made on constituent applications, in part because often only the most basic hooks are available if at all, and in part because the same demands are made of more or less all constituents, which exacerbates the previous problem. In the end, the resulting heterogeneous system always feels to the user just like what it is: a confederation of largely unrelated parts. In short, this approach enjoys an immediate, immense gain in functionality at the start, but suffers from the at times ungainly interface of presenting a set of relatively loosely coupled parts.

On the other hand, inaugurating a new framework, as Brown notes in the quote above, demands a large amount of reimplementation at worst, adaptation at best, of common tools. In the Multivalent case, one is reluctant to use some platform-specific tool and give up the otherwise platform-independent quality of Java as it is the rare tool that can be found on all platforms, with the same interface. Large, site-specific tools, such as data bases and OCR engines, can be integrated with a proxy wrapper that adheres to the framework and communicates in a non-portable way with the main tool. Core functionality of the framework, however, such as a function that computes the differences between two files, must be reimplemented. This is potentially a large amount of work, though it is a one-time cost.

Of course, if tools are built within the framework from the start, one enjoys the benefits of both approaches. The Multivalent model aims to provide an attractive platform for such new tools.

7.6 Fundamental API vs. Open Source: Netscape 5, TkMan/Tk Text Widget

The GNU Project's definition of free software [Stallman 1985], also called open source, means that the software's source code is available to be debugged and improved outside of the company or chief author (only incidentally is the software usually free of charge). Netscape Navigator version five (also known as Mozilla) [Mozilla 1998] and the Tcl/Tk scripting language and graphical user interface toolkit [Ousterhout 1994] are two examples: Netscape Corp. and Scriptics, respectively, are the chief developers but distribute the source code so that a large audience over the Internet can help debug and improve the software.

Given the source code of a product written in a general purpose programming language, one can of course customize the software as much as desired. When there is a well defined goal (such as the GNU Project's goal of duplicating UNIX utilities), open source

development can work well. But in a system open to experimentation that by nature does not converge, distributing the results is problematic: Either one's work permanently diverges from the mainstream code line, or it is synchronized with the mainstream and composed with other work. The former case obviously suffers in that future improvements are foregone or integrated with increasingly greater effort, and it also imposes an enormous burden on the end user who must use that divergent and probably soon-obsolete code line to take advantage of the local customization. The latter case in which code is resynchronized suffers from the problem that probably only broadly useful extensions will be accepted by the chief of synchronization in order to avoid code bloat, for the sake of preserving code maintainability and reasonable runtime resource requirements. If the customization is rejected from the mainstream, one can distribute a patch.

While the above scenario is difficult for any software, it creates an impossible situation for a web browser like Netscape Navigator. If we assume that maintainers of libraries have specialized the browsers to work better with their collections, then web surfers, now accustomed to browsing all pages within the same environment, must first download the special browser for that collection before viewing that collection. (A cross-platform browser such as one written in Java only saves a compilation step.) What's needed is a browser that can be adapted modularly. Yet as applications demand more and more capabilities, and the browser defines more protocols to integrate modules and prevent conflicts among them, the result to a first approximation is the Multivalent architecture.

The difficulty of sharing improvements to open source software was experienced firsthand by the present author with TkMan [Phelps 1994] and Tcl/Tk. TkMan needed to patch the Tk text widget [Ousterhout 1993] to introduce "elided" text—that is, text that remains in the widget, but that can be "switched" on or off (ignored or not) during formatting and painting. But first a background sketch of that widget.

Tk's text widget displays text in multiple fonts and colors, embedded images, embedded windows, and underlining, margins, tabs, justification, and other effects. Tags (similar to Multivalent spans) over a continuous span of characters control special properties such as font, color, underlining, and event bindings. Marks name a point between two characters robust to document editing. Like Emacs, the text widget models text weakly as a series of lines. Programmatic introspection and control of the text widget are excellent. Not an editor meant for the end user out of the box, it provides an API to C/C++ and various scripting languages, especially Tcl. All widget entities can be queried, created and destroyed, and (re)configured at any time. Any character or line can report its bounding box, although for performance reasons lines not shown on the screen have null geometric extent. Windows, images, marks and tags can all enumerate their instances, which can be reconfigured or mapped into specific locations in the text widget, where further editing can take place.

Beyond its powerful built-in capabilities, the text widget is difficult to extend in unanticipated ways. It cannot be extended within the interface it supplies, but its source code is available. Its C code implementation is complex and not intended to be examined

by third parties. And once the text widget source code is understood and an innovation achieved, it is difficult to distribute the innovation, partly because the patch will likely conflict with any other patches, and partly because any patch to the core system requires a (partial) recompilation, a situation exacerbated on Macintosh and Windows platforms where the general user is unlikely to have the tools or the technical expertise.

For instance, a patch used by TkMan adds "elided" text which is useful for collapsible outlines and the prototype for Notemarks (see Section 6.3.3). Because outlines and Notemarks were thought compellingly useful, the new version of TkMan incorporating these features made the elided text patch mandatory. Otherwise, it was thought, users would weigh the labor of patching and recompiling against unfamiliar benefits, bypass elided text, and never experience the innovations. Experience has shown that patching and recompiling is the least enjoyed aspect of installation. Unfortunately, the only way to make it easier is for the patch to be accepted into the core Tk text widget, and an appeal for this has fallen on deaf ears. Moreover, although the text widget has remained static for years, the elide text patch has conflicted with both of the two other known patches to it, one by a third party, the other with the main distribution when Unicode support was incorporated, thus requiring three slightly different patches, each specifically accommodating the other known party.

Tcl/Tk, as a high level scripting language and rich user interface toolkit, makes it easy to write programs whose needs lie within the features they provide. Extending the C code in which the Tk widgets are written, while not often necessary, is very difficult and subsequently it is extremely awkward to distribute the patches. In other words, it makes the easy things easy, the medium things easy, but the hard things perhaps extremely difficult. The Multivalent framework, in contrast, concentrates on making everything possible, through presently the bare protocols could use wrappers to provide higher level interfaces to ease programming. In other words, it makes the easy things medium, the medium things medium, the hard things medium or hard things proportional to their intrinsic difficulty. It would be straightforward to implement the features of TkMan in Multivalent, nearly impossible the other way around.

7.7 Cross-format vs. Single Format: IDVI/TeX DVI

IDVI [Dickie 1996] is a browser for TeX [Knuth 1984] device-independent (DVI) files. DVI files are similar to scanned page images insofar as they describe the appearance of the pages of a document in such detail that they can be displayed or printed identically on any system. IDVI is a Java applet that accurately displays DVI files with dynamically loaded TeX-specific fonts, encapsulated PostScript images (which have an embedded bitmap representation), colored text, hypertext links, text hiding and outlines, magnification, and internal Java applets. It does not allow text selection or have annotation capabilities.

One advantage of the Multivalent approach is its dynamic extensibility and composition. The author of a TeX document can insert arbitrary instructions into a DVI file by means of the escape command \special. If a DVI viewer does not recognize the special, it ignores it. It is awkward to introduce new specials into documents because it requires updating a large number of viewers, maintained by numerous authors on numerous platforms, a situation which leads to balkanized features with viewers supporting varying subsets of specials. Even if the TeX community standardized on IDVI as the common browser, which would not be unreasonable since as a Java applet it runs on all popular platforms and it is in many ways the best DVI viewer, IDVI would need be extended to easily incorporate code for new specials and to define interfaces so that new specials have access to any part of the document they may need (that is, all parts of the document) and do not conflict with each other, two requirements that are the foundation of the Multivalent model.

A second advantage of the Multivalent approach is its straightforward assimilation of multiple document formats. Though popular among mathematicians and other technical communities, DVI has limited popularity compared to numerous other formats such as PostScript or HTML. Thus any DVI system is likely to receive a limited amount of resources. Furthermore, resources spent on DVI do not benefit other formats, and conversely resources spent on other formats do not benefit DVI. In the Multivalent approach, DVI would be supported with a relatively small amount of DVI-specific effort, specifically parsing the format and constructing a corresponding hierarchical tree and receive the benefit of the great majority of the work put into the system, both in common infrastructure and in extension behaviors.

By contrast, when a Multivalent DVI media adaptor is written, it will immediately benefit—"for free"—from selections, annotations, lenses, speed reading, and almost all of the functionality available to other document formats. Likewise, almost any extension written for use in DVI flows back for immediate use by other document formats. As improvements are made in infrastructure or behaviors, the benefits are seen by all document formats because they share the same behaviors. In non-Multivalent systems, each additional feature would need to be built from scratch or at least extensively adapted for DVI, and as the feature is improved, its derivatives would need to track this fact and be updated.

8 Implementation Experience and Future Work

Unsurprisingly, implementation experience proved vital for fleshing out the architecture. The first part of this chapter recounts surprises that in some cases led to making the framework more flexible and in other cases more structured.

Work described in this dissertation is an ongoing project. Experience with real users will undoubtedly make the system enormously better and, hopefully, point out applications that could be uniquely served with this technology. The second part of this chapter describes foreseeable short-term, medium-term and long-term work.

The implementation of the work described is available via the web page at http://www.cs.berkeley.edu/~phelps/Multivalent/. It is free and unencumbered for all uses.

8.1 Implementation Experience

Not only was it possible to abstract over different document types, but the specialization for a particular type was surprisingly small. If the system becomes popular and third parties write behaviors, it will be important to make sure they target the abstract document as much as possible, not individual document formats, so that the system does not begin to resemble the current situation of different systems for every slightly different document type.

The basic architectural conception proved quite stable over time. An initial idea that behaviors would ordinarily communicate directly with one other (as dynamically joined in a "type graph") was scrapped along with the initial monolithic prototype in favor of the second, protocol- and behavior-based implementation. Thereafter, almost no changes were made in protocol method signatures—the information passing contract between the core system and individual behaviors—as a wide variety of behaviors were implemented.

In fact, the architecture proved more powerful than at first appreciated. The initial implementation of lenses was moved from an ordinary behavior into the core, but the simple addition of a mechanism for dynamically sharing behavior instances proved sufficient to move them back out. This dynamic sharing also proved sufficient to synchronize time-based media, that is, to introduce the concept of time into the model without modifying the core. Annotations are presently implemented as simple strings due to memory space concerns, but on second consideration it was realized that the document tree is quite lightweight, and annotations will in the future have their own trees as Notes do presently, to enjoy multi-font, linebroken text that can itself be annotated.

In a few cases, notably the selection, spans, and attributes of the graphics context, which in fact could be implemented only with system primitives, their near-universal use led to incorporating corresponding interfaces into the core to provide greater convenience for the programmer.

8.2 Future Work

The mandate for the future is to prepare the system for widespread use, then to continuously improve it, informed by users. Reaching a first general public release requires solidifying the current implementation: improving usability (user interface and robustness), and fixing and finishing components (such as HTML). If the system gains popularity, it is expected that users will want more document formats (XML, RTF, PDF), and behavior writers will want an expanded set of behaviors in the basic system (not hardcoded into the core, but distributed with it so as to guarantee availability). Finally, with a robust and powerful platform in place, most interesting will be experimentation with potentially useful document ideas, many of which are awkward or impossible to integrate well and then distribute with previous systems.

8.2.1 Requirements for Widespread Distribution

It is critical that the user interface be as usable as possible. Simple use should be no more difficult than clicking in a web browser, a skill mastered (or able to be mastered) by everyone. Taking advantage of more advanced behaviors will require more learning on the part of the user, a process to be facilitated by the system. Although the specifics of a behavior's interface, as authored by a third party, lie beyond the core system's reach, the

system will need to standardize a way to report which behaviors are available and to access the behavior's documentation. The current system's help-on-menu-item is a first step.

The user interface needs to be updated. The code was developed with Java version 1.0.2, with its anemic user interface, lacking among other essentials radio menu items and pop-up menus. Introduced with Java 1.1 was the "Swing" user interface widget set, with a full set of modern user interface components. Numerous interface improvements are now within easy reach, such as a pop-up menu for spans that permits deleting the span or editing its properties. As well, the rest of the code needs to be updated for Java 1.1 and soon Java 1.2 with the latter's rich set of advanced functionality, including excellent two-dimensional graphics that could directly improve document appearance.

Aside from bug fixes, the system will need to be made robust to hostile environments, including corrupted files and bad network connections. Real users are sure to discover every way the system can fail. The system also needs to be easy to install; fortunately with Java one can easily generate and distribute effectively executable code for all popular platforms.

The media adaptor for HTML needs to be finished to handle forms and floating figures. The hub document format was designed as a trivially parsable SGML (all tags balanced and well nested), which is essentially the same as XML, and it would be worthwhile investing the small amount of effort needed to bring the implementation into conformance with the XML specification.

8.2.2 Second Generation Work

Some protocols can be usefully further opened. Restore instantiates behaviors listed by name in the hub document; it could be useful to recognize a "behavior substitution mapping" that can dynamically replace the named behavior with an improved or alternative behavior. Likewise, in building the user interface from information in the user interface tree, the user may prefer improved or different styles of user interface widget, and the system should allow substitutions here as well.

While the basic algorithms are scalable and the system performs responsively, future performance profiling will undoubtedly boost performance more. Although memory is inexpensive, memory use tuning will better handle larger documents and more of them at once.

As the user adds more and more behaviors to the standard set, some hooks into the control flow may become hot spots. For instance, among those behaviors that will logically register interest in the tree root where they can examine every keypress are those for character insertion, accent specification, abbreviation expansion, auto-correction from common typographical errors, spelling correction, and "smart" quotes.

Again, the basic algorithms are scalable (often through the use of hierarchy), and they will be tuned as needed.

Among the major new behaviors to be introduced in the near future, media adaptors for RTF (which almost all word processors can generate) and PDF (to which PostScript can be converted)—when added to the existing HTML, ASCII, and scanned page images—will cover the most popular document formats in use today. Future media adaptors will probably include PowerPoint, (direct) PostScript, Texinfo (which is not very popular but is straightforward to support), and TeX DVI.

Although the Multivalent model emphasizes semantic layers of information, trees should support visual overlays. Individual behaviors could implement this without system support by registering interest in the root and always draw, but the system should provide data structures that efficiently support visual content running along the extent of a long document.

It can be useful to save, as much as possible, a Multivalent document in an existing, "reduced" format. Given the cross product of behaviors and formats, the system must either encode knowledge of behaviors into formats or of formats into behaviors. As neither strategy is scalable, a hybrid strategy seems likely. For the bulk of the translation, the media adaptor for desired save format walks the document tree translating behaviors it recognizes. For common cases (such as italics and simple hyperlinks), a shared set of known behavior types makes this a feasible scheme. New behavior types not in the common set can offer custom translations to various formats. For example, a Note may be able to render a facsimile of itself in HTML using layers. It would be useful for all media adaptors to generate a simple ASCII representation of their media type for use by full text indexers.

TeX is widely acclaimed for its linebreaking and mathematical layout of the highest quality. Though TeX itself is a batch formatter, several systems, including doc [Calder and Linton 1992], VorTeX [Chen *et alia* 1988], and Textures [Blue Sky 1990], provide an interactive interface. Although providing TeX typesetting for HTML messages could be considered overkill, this would be a no-cost side benefit of work targeted primarily at, say, word processing applications.

A scripting language has proven a useful, high level means of control in many systems, and one will probably be added, almost certainly chosen from existing languages such as JavaScript, Tcl, or Scheme. The system is believed to supply a sufficient interface for a scripting language, and it is hoped that third parties interface more than one.

One useful potential behavior could maintain historical version trees of documents (such as web pages), so a frequented page would always be available even if the network is down and, coupled with a differencing engine, the changes between the current page content and the last time the page was visited could be displayed, thus showing what is new to you the individual user, without relying on server support.

Layout based on physical layout trees needs to be added distinct from the logical document tree. To a considerable degree, physical extent and logical content are coincident, but they diverge in the cases common in printing of splitting a structural unit such as a paragraph or even a chapter across pages, and as well separate physical trees are needed to support multiple views.

8.2.3 Research Issues

It will be interesting to see how well the Multivalent model generalizes outside of documents proper. Some work with time-based media [Matusik 1998] has synchronized audio and video with other document elements such as spans (hyperlinks, highlights) and lenses (for subtitles in multiple languages). It is believed that an introduction of the concept of a *dimension* into the core architectural would be a key step in augmenting the two dimensional document plane to 3-D (x, y, z dimensions), geographic information systems (latitude, longitude, height; with layers), video (x, y, time dimensions), audio (time), and so on. But more work is needed to determine if such systems significantly benefit from a Multivalent implementation. Even in areas that may or may not be usefully adapted to the Multivalent model per se, users can benefit from a document system adaptable to specialized environments, as was demonstrated with the Henry system [Silva 1994], a hypermedia front end to a CAD system.

Much attention has been paid to the security of Java applets. Since a significant advantage of the system is its seamless integration of third-party behaviors, behaviors have the same security risk as applets. However, it is much more difficult if not impossible to "sandbox" behaviors in the way that applets are. Applets have relatively little interaction with the rest of the system—they are isolated rectangles that on occasion communicate with other applets on the same page—whereas behaviors have broad access to and control over the document. Existing applets and their security guarantees can be supported, but for behaviors a degree of trust, perhaps based on cryptographic signatures, will be required at least for the foreseeable future.

Documents and user interfaces share much common infrastructure: layout, tables, specialized behaviors, events, painting. And several systems unifying documents and user interface widgets can be found in the literature [Bier and Goodisman 1990, Bier 1991, Tang and Linton 1994]. The lack of such a unification in current web browsers, which use user interface widgets native to the platform, results in an jarring aesthetic dissonance between elements of forms (with their text entry, menu, button widgets) and the rest of the document. The current document services would need only small specialization, and the advantage of a homogeneous and equally power user interface set will be weighed against the immediate availability of Java's now-complete user interface widget set.

In the less immediate future, it will be interesting to see how well the Multivalent model works in different devices. Its extreme customizability and potentially small footprint

(just a small core with behaviors loaded on demand) should make a good match for small, mobile devices, whether networked (wirelessly) or not.

9 References

Adobe Systems. 1998. Acrobat 3.0. http://www.adobe.com/special/productfinder/acrobat-advanced-all.html

Adobe Systems. November 1996. Portable Document Format Reference Manual, Version 1.2. http://www.adobe.com/supportservice/devrelations/PDFS/TN/PDFSPEC.PDF

Apple Computer. 1995. *OpenDoc Programmer's Guide*. Addison-Wesley.

Apple Computer. October 1993. OpenDoc Technical Summary.

Steven C. Bagley and Gary E. Kopec. December 1994. Editing Images of Text. *Communications of the Association for Computing Machinery*, pages 63-72.

Stephen A. Barney, editor. 1991. *Annotation and Its Texts*. Oxford University Press.

Eric A. Bier and Aaron Goodisman. 1990. Documents as User Interfaces. *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP '90)*, Cambridge University Press, pages 249-262.

Eric A. Bier. November 1991. EmbeddedButtons: Documents as User Interfaces. *Proceedings of User Interface Software and Technology (UIST '91)*, Hilton Head, South Carolina, pages 45-53.

Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton and Thomas Baudel. April 1994. A Taxonomy of See-Through Tools. *Proceedings of CHI '94*, Boston, MA, pages 358-364.

Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton and Tony D. DeRose. August 1993. Toolglass and Magic Lenses: The See-Through Interface. *Proceedings of SIGGRAPH '93*, Anaheim, California, pages 73-80.

Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. May 1998. Cascading Style Sheets, level 2 CSS2 Specification. World Wide Web Consortium Recommendation, http://www.w3.org/TR/REC-CSS2/.

Blue Sky Research. Textures. http://www.bluesky.com/

Andrea Bozzi and Sylvie Calabretto. September 1997. The Digital Library and Computational Philology: The BAMBI Project. *First European Conference on Digital Libraries*, pages 269-285.

Tim Bray, Jean Paoli and C.M. Sperberg-McQueen. 1998. Extensible Markup Language (XML), http://www.w3.org/TR/1998/REC-xml-19980210.

Kraig Brockschmidt. 1995. *Inside OLE 2*. Microsoft Press.

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

Marc H. Brown. November 1995. Browsing the Web with a Mail/News Reader. *Proceedings of the ACM Symposium on User Interface and Software and technology (USIT 95)*, Pittsburgh, pages 197-199.

P.J. Brown. November 1992. UNIX Guide: Lessons from Ten Years' Development. *Proceedings of the ACM Conference on Hypertext (ECHT '92)*, Milano, Italy, pages 63-80.

M. Cecelia Buchanan and Polle T. Zellweger. November 1992. Specifying Temporal Behavior in Hypermedia Documents. Proceedings of the ACM Conference on Hypertext (ECHT '92), Milan, Italy, pages 262-271.

Vannevar Bush. 1945. As We May Think. *Atlantic Monthly*.

Paul Calder and Mark Linton. 1992. The Object-Oriented Implementation of a Document Editor. *Proceedings of OOPSLA '92*, pages 154-165.

Donald D. Chamberlin, Helmut F. Hasselmeier, Allen W. Luniewski, Dieter P. Paris, Bradford W. Wade, and Mitch L. Zolliker. 1987. Quill, An Extensible System for Editing Documents of Mixed Type. *Proceedings of the 21st Hawaii International Conference on System Sciences*, IEEE Computer Society Press, pages 317-326.

Pehong Chen, Michael A. Harrison, and Ikuo Minakata. December 1998. Incremental Document Formatting, Proceedings of the ACM Conference on Document Processing Systems, Santa Fe, New Mexico, pages 93-100.

James Clark. SP. http://www.jamesclark.com/.

Daniel S. Connelly, Beth Paddock and Rebecca Rice. March 1995. The XDOC Data Format, Version 3.0. Xerox Corporation.

Garth A. Dickie. 1996. IDVI. http://www.geom.umn.edu/java/idvi/

D.C. Engelbart. October 1962. Augmenting Human Intellect: A Conceptual Framework. http://www.histech.rwth-aachen.de/www/quellen/engelbart/ahi62index.html

Michael J. Fischer and Richard E. Ladner. June 1979. Data Structures for Efficient Implementation of Sticky Pointers in Text Editors (Extended Abstract). University of Washington Department of Computer Science, Technical Report 79-06-08.

A. Fountain, W. Hall, I. Heath and H. David. 1990. Microcosm: An Open Model for Hypermedia with Dynamic Linking. *Proceedings of ECHT '90*, Association for Computing Machinery.

David Fox. 1998. Composing Magic Lenses. *Proceedings of CHI '98*. Los Angeles.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. 1995. *Design Patterns*. Addison-Wesley.

Charles F. Goldfarb. 1990. *The SGML Handbook*. Clarendon Press.

James Gosling. October 1984. An Editor-Based User Interface Toolkit. *Proceedings of the First International Conference on Text Processing Systems, Dublin*, Ireland, pages 126-132.

James Gosling, Bill Joy and Guy Steele. 1996. The Java Language Specification. Addison-Wesley.

Susan L. Graham, Michael A. Harrison and Ethan V. Munson. 1992. The Proteus Presentation System. Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, pages 130-138.

Frank G. Halasz, Thomas P. Moran and Randall H. Trigg. April 1997. NoteCards in a Nutshell.. *Proceedings of the ACM CHI + GI Conference*, pages 1-9.

Frank G. Halasz. July 1988. Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the Association for Computing Machinery*, pages 836-852.

Barry W. Holtz, editor. 1984. *Back to the Sources: Reading the Classic Jewish Texts*. Summit Books, New York. See especially pages 140-141.

Jeremy Hylton. May 1994. Text-Image Maps for Document Delivery on the Web. *Proceedings of the First HTML+ Workshop International Conference on the World-Wide Web*, http://ltt-www.lcs.mit.edu/ltt-www/Papers/jerhy-www94.html.

International Organization for Standardization. 1992. Hypermedia/Time-based Structuring Language: HyTime. ISO/IEC 10744

ISO/IEC 10179:1996. 1996. Document Style Semantics and Specification Language (DSSSL)

Insight Development. 1998. Hot Off the Web. http://www.hotofftheweb.com/.

Internet Mute. 1998. interMute. http://www.intermute.com

Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press.

Donald E. Knuth. 1984. *The TeXbook*. Addison-Wesley.

Daryl T. Lawton and Ian E. Smith. November 1993. The Knowledge Weasel Hypermedia Annotation System. *Proceedings of the Hypertext '93*, pages 106-117.

David M. Levy and Catherine C. Marshall. April 1995. Going Digital: A Look at Assumptions Underlying Digital Libraries. *Communications of the Association for Computing Machinery*, pages 77-84.

Bil Lewis, Dan LaLiberte, and Richard Stallman. June 1995. *GNU Emacs Lisp Reference Manual, Edition 2.4*. Free Software Foundation, Cambridge, MA.

Francis Li. 1998. Private communication.

Mainstay. MarkUp 2.0. http://www.mstay.com/mu20.html

Udi Manber. December 1997. Creating a Personal Web Notebook. *Proceedings of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA.

Catherine C. Marshall. July 1997. Annotation: From Paper Books to the Digital Library. *Proceedings of The Second ACM Conference on Digital Libraries*, Philadelphia, Pennsylvania, pages 23-26.

Catherine C. Marshall. June 1998. Towards an Ecology of Hypertext Annotation. *Proceedings of Hypertext '98*, Pittsburgh, PA.

Wojciech Matusik. 1998. Structured Time-based Media in the Multivalent Framework. Personal communication.

Vance Maverick. January 1998. Presentation by Tree Transformation, Technical Report 97-947. Computer Science Division, University of California, Berkeley, http://sunsite.berkeley.edu/Dienst/UI/2.0/Describe/ncstrl.ucb%2fCSD-97-947.

Owen McGrath. 1996. Private communication.

Microsoft. Dynamic HTML. September 1997. http://www.microsoft.com/workshop/author/dhtml/dhtmlovw.asp

Microsoft. Visual Basic.

The Mozilla Organization, http://www.mozilla.org.

Ethan Vincent Munson. December 1994. Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment. Ph.D. dissertation, Computer Science Division, University of California, Berkeley.

National Center for Supercomputing Applications (NCSA). Mosaic. http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html

Theodor H. Nelson. 1974/1987. *Computer Lib/Dream Machines*.

Netscape Communications. 1997. Navigator 4.0. http://home.netscape.com/

Netscape Communications. 1998. Dynamic HTML. http://developer.netscape.com/tech/dynhtml/index.html

Steven R. Newcomb, Neill A. Kipp and Victoria T. Newcomb. November 1991. The HyTime Hypermedia/Time-based Document Structuring Language. *Communications of the Association for Computing Machinery*, pages 67-83.

Vincent Quint and Irène Vatton. 1986. Grif: An Interactive System for Structured Document Manipulation. *Proceedings of the International Conference on Text Processing and Document Manipulation*, University of Nottingham, pages 200-213.

Kenton O'Hara and Abigail Sellen. March 1997. A Comparison of Reading Paper and On-Line Documents. Proceedings of the Conference on Human Factors in Computing Systems (CHI '97), Atlanta, Georgia, pages 335-342.

John Ousterhout. March-April 1993. Hypergraphics and Hypertext in Tk. *The X Resource*, 1(5):74-81.

John K. Ousterhout. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley.

Dave Raggett, Arnaud Le Hors, and Ian Jacobs. April 1998. HTML 4.0 Specification. World Wide Web Consortium Recommendation, http://www.w3.org/TR/REC-html40/.

Thomas A. Phelps. 1994. TkMan: A Man Born Again. *The X Resource*, 1(10):33-46.

Thomas A. Phelps and Robert Wilensky. September 1997. Multivalent Annotations. *Proceedings of the First European Conference on Digital Libraries*, Pisa, Italy, pages287-303.

Thomas A. Phelps and Robert Wilensky. March 1996. Multivalent Documents: Architecture and Applications. *Proceedings of the First ACM International Conference on Digital Libraries*, Bethesda, Maryland, pages 20-23.

Thomas A. Phelps and Robert Wilensky. January 1996. Multivalent Documents: Inducing Structure and Behaviors in Online Digital Documents. *Proceedings of the 29th Hawaii International Conference on System Sciences*, Maui, Hawaii.

Brian K. Reid. 1980. Scribe: A Document Specification Langauage and its Compiler, CMU-CS-81-100, Carnegie-Mellon University.

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

Robert Sanderson. 1998. Linking Past and Future; an Application of Next Generation Computing Technology for Medieval Manuscript Editions, Ph.D. thesis proposal, http://gondolin.hist.liv.ac.uk/~azaroth/university/proposal.html

Mário J. Silva. 1994. Active Documentation for VLSI Design. Ph.D. dissertation, Computer Science Division, University of California, Berkeley.

Richard M. Stallman. June 1981. EMACS: The Extensible, Customizable, Self-documenting Display Editor. *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, Oregon, pages 147-156.

Richard M. Stallman. July 1997. *GNU Emacs Manual, Thirteenth Edition*. Free Software Foundation.

Richard M. Stallman. 1985. The GNU Manifesto. http://www.gnu.org/gnu/manifesto.html.

Steven H. Tang and Mark A. Linton. 1994. Blending Structured Graphics and Layout. Proceedings of User Interface Software and Technology (UIST '94), Marina del Rey, California, pages 167-174.

Tenax Software Engineering. Vortex. http://www.vallier.com/tenax/vortex.html

Douglas B. Terry and Donald G. Baker. June 1990. Active Tioga Documents. Technical Report CSL-90-6, Xerox Corporation, Palo Alto Research Center (PARC).

Frank Wm. Tompa, G. Elizabeth Blake and Darrell R. Raymond. 1993. Hypertext by Link-Resolving Components. *Proceedings of Hypertext '93*, Seattle, Washington, pages 118-130.

R. H. Trigg. November 1983. A Network-Based Approach to Text Handling for the Online Scientific Community. Ph.D. Thesis, Dept. of Computer Science, University of Maryland (University Microfilms #8429934).

Irène Vatton, Ramzi Guétari, José Kahan, and Vincent Quint. 1996. Amaya. World Wide Web Consortium, http://w3c.org/Amaya/.

Larry Wall. patch.

Brent Welch and Steve Uhler. July 1996. Tcl/Tk HTML Tools. *Proceedings of the Fourth Annual Tcl/Tk Workshop*, Monterey, California, USENIX Association, pages 173-182.

Robert Wilensky. April 1995. UC Berkeley's Digital Library Project. *Communications of the Association for Computing Machinary*, page 60.

Robert Wilensky. May 1996. Toward Work-Centered Digital Information Services. *IEEE Computer Special Issue on Digital Libraries*.

Lauren Wood et alia. July 1998. Level 1 Document Object Model Specification. http://www.w3.org/TR/WD-DOM/

World Wide Web Consortium. W3C Core Styles. http://www.w3c.org/StyleSheets/Core/