UNIVERSITY OF CALIFORNIA,
IRVINE

An Analysis of the Hypertext Versioning Domain

DISSERTATION

submitted in partial satisfaction of the degree requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Emmet James Whitehead, Jr.

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

The dissertation of Emmet James Whitehead, Jr. is approved
and is acceptable in quality and form
for publication on microfilm:

____

____

____
Committee Chair

University of California, Irvine
2000

Table of Contents

List of Figures .... **vi**
List of Tables .... **viii**
Acknowledgements .... **ix**
Curriculum Vitae .... **xi**
Abstract of the Dissertation .... **xv**

Chapter 1 – Introduction **1**
1.1 Hypertext, and Hypertext Versioning 1
1.2 Domain Analysis 7
1.3 Domain Analysis of Hypertext Versioning 11

Chapter 2 – Domain Terminology **16**
2.1 General Terms 16
2.2 Hypertext 17
2.2.1 Work 18
2.2.2 Anchor 27
2.2.3 Link 30
2.3 Version 35
2.3.1 State-based and Change-based Versioning 36
2.3.2 Variant 38
2.3.3 Versioned Object 41

Chapter 3 – Taxonomy of Hypertext Versioning Systems **44**
3.1 Introduction 44
3.2 Versioning for Reference Permanence 44
3.3 Versioned Data, Unversioned Structure 45
3.4 Composite-based Systems 46
3.4.1 PIE, a Change-Oriented Composite-Based System 47
3.5 Web Versioning 48
3.6 Versioning for Open Hypertext 49

Chapter 4 – Containment **51**
4.1 Introduction 51
4.2 Basic Static Containment 53
4.2.1 Independence of Mathematic Set Properties 58
4.2.2 Deletion Semantics 59
4.3 Relationship Abstraction Layers 61
4.4 Common Definitions of Inclusion and Reference Containment 64
4.5 Links, Containment Relationships, and Containers 65
4.6 Dynamic Containment 66
4.7 Advanced Containment Semantics 67

Chapter 5 – Address and Name Spaces **68**
5.1 Centralized Assignment and Resolution 70
5.2 Decentralized Assignment and Resolution 71

Chapter 6 – Modeling System Data Models 74
6.1 Modeling Primitives 77
6.1.1 Entities 77
6.1.2 Relationships.. 78
6.2 Data Modeling Examples.. 79
6.3 Relation of Versioning Scenarios to Data Model.. 83

Chapter 7 – Domain Reference Requirements.. 96
7.1 Data Versioning 96
7.1.1 The history of objects must be persistently stored. 96
7.1.2 Mutability of primary state and metadata for object revisions must be supported............... 98
7.1.3 Versioned and non-versioned objects can coexist.. 99
7.1.4 All content types must be versionable 100
7.1.5 A mechanism must exist for giving a human readable name to a single revision.............. 101
7.1.6 Revisions and versioned objects can be removed. 101
7.2 Stability of References.. 102
7.3 Change Aggregation Support.. 102
7.4 Link and Structure Versioning.. 103
7.4.1 It must be possible to version links. 104
7.4.2 It must be possible to version structure. 104
7.4.3 It must be possible to link to a specific revision. 105
7.5 Variant Support.. 105
7.6 Collaboration Support.. 106
7.7 Navigation in the Versioned Space.. 107
7.8 Searching 108
7.9 Visualizing the Versioned Space 109
7.10 Traceability.. 110
7.11 Goals for User Interaction.. 111
7.12 Goals for Tool Interaction.. 111
7.13 Goals for Interactions with an External Repository.. 113
7.14 Namespace Interactions 114
7.15 Maintaining Consistent Structures.. 115

Chapter 8 – Design Spaces and Tradeoffs Associated with Domain Requirements 116
8.1 Object Organization.. 116
8.2 Data Versioning 117
8.2.1 Persistent Storage of Revision Histories.. 117
8.2.2 Versioning Dynamic Content and Dynamic Links 126
8.2.3 Revision Mutability, and Immutability.. 127
8.2.4 Coexistence of Versioned and Unversioned Objects 128
8.2.5 Versioning of all Content Types 129
8.2.6 Revision Naming 131
8.2.7 Removing revisions and versioned objects.. 132
8.3 Stability of References.. 132
8.4 Change Aggregation Support.. 134
8.5 Link and Structure Versioning.. 137
8.5.1 Link versioning. 137
8.5.2 Structure versioning. 141
8.5.3 Linking to a Specific Revision.. 155
8.6 Variant Support.. 156
8.6.1 Individual Object Variation 156
8.6.2 Compound Object Variation 159
8.7 Collaboration Support.. 160
8.7.1 Concurrency Control.. 161
8.7.2 Workspaces.. 167
8.7.3 Merging Hypertext Networks 169

8.8 Navigation in the Versioned Space 171
8.9 Searching 172
8.10 Visualizing the Versioned Space 173
8.11 Traceability 174
8.12 User Interaction 175
8.13 Tool Interaction 177
8.14 Interaction with an External Repository 180
8.15 Namespace Interactions 181
8.16 Maintaining Consistent Structures 181

Chapter 9 – Applying the Domain Model 183
9.1 WebDAV and DeltaV 184
9.1.1 Requirements 185
9.1.2 Design Choices 192
9.2 Chimera 199
9.2.1 Requirements 201
9.2.2 Design Choices 205

Chapter 10 – Related Work 213
Chapter 11 – Future Work 217
Chapter 12 – Contributions and Conclusion 221
References 225

List of Figures

Figure 1 – An intellectual work, at the conceptual level and the computer representation level. 24

Figure 2 – The conceptual model of an anchor, and its computer representation. 29

Figure 3 – The conceptual model of a link, and its representation within a computer. 33

Figure 4 – A revision history shown with state and change orientation. 38

Figure 5 – An intellectual work, with revisions and variants recorded, and contained within a versioned object. 43

Figure 6 – The five primary types of containment, plus one hybrid containment type. 57

Figure 7 – A container at three different layers of abstraction.. 63

Figure 8 – Data model of the Dexter hypertext reference model. 80

Figure 9 – Data model of the Intermedia system. 81

Figure 10 – Data model of the NoteCards system. 82

Figure 11 – Overview of the collaborative work scenario. 87

Figure 12 – Data model of Neptune/Hypertext Abstract Machine (HAM) 88

Figure 13 – Common collaborative work scenario, as represented by Neptune. 89

Figure 14 – Data model of the HyperPro system. 90

Figure 15 – Object inheritance diagram for the HyperPro system.. 90

Figure 16 – Common collaborative work scenario, as represented by HyperPro. 91

Figure 17 – Data model of the CoVer system. 92

Figure 18 – Common collaborative work scenario, as represented by CoVer. 93

Figure 19 – Data model of the HyperProp/Nested Context Model. 94

Figure 20 – Common collaborative work scenario, as represented by HyperProp/NCM. 95

Figure 21 – State-based design choices for representing a linear version history...................................... 123

Figure 22 – The state-based versioning design space.. 124

Figure 23 – Indirect referential containment.. 144

Figure 24 – Link proliferation in the absence of revision selection rules. 146

Figure 25 – A revision selection rule on the container chooses the revision for each link endpoint; link endpoints are versioned objects. 146

Figure 26 – Containment diagram showing versioned structure using unversioned links, and a revision selection rule on each structure collection revision. 149

Figure 27 – Containment diagram showing versioned structure using unversioned links that employ indirect referential containment for their endpoints. 151

Figure 28 – Two revisions of structure container C, each containing an unversioned link, *l*, and two contained work revisions, the endpoints of the link are the same revisions, since the link uses indirect referential containment. 152

Figure 29 – Containment diagram showing versioned structure using versioned links. Structure containers and links use containment where the revision selection rule is on the containment arc. 154

Figure 30 – An example of structure versioning using versioned links. The structure container, C, contains one revision from versioned objects A and B, and one link revision from versioned link *l*. Each containment arc has its own revision selection rule. 154

Figure 31 – The data model of the WebDAV extensions to HTTP. 185

Figure 32 – Data model of the DeltaV versioning and configuration management extensions to WebDAV.. 194

List of Tables

Table 1 – Hypertext systems, and their term for linkable information, and link endpoint........................... 26

Table 2 – Static containment characteristics for composite-based hypertext versioning systems, the Dexter reference architecture, and the DMA 1.0 standard 58

Table 3 – Interactions between containment type, and mathematic set properties of containers................. 59

Table 4 – Variations in delete semantics for different containment types. 61

Table 5 – Characteristics of inclusion and reference containment... 65

Table 6 – Design options for recording the history of objects. 125

Table 7 – Design options for recording link history. 140

Table 8 – Concurrency control techniques organized by how long they maintain variants created by individual authors. 167

Acknowledgements

Throughout my graduate career, my advisor Richard Taylor has been an enormous influence. By developing my critical analysis skills, guiding my understanding of how to perform research, providing varied research projects early in my career spanning user interface toolkits and software architecture, and by exercising a laser-sharp ability to identify areas requiring further work, Professor Taylor's influence on me, and this dissertation, have been profound.

David Rosenblum, Mark Ackerman, and John King all enriched the dissertation with their comments and questions, and motivated me with their interest and encouragement.

I am very grateful to my wife, Julia, for your love, warmth, and encouragement during the long slog of writing. My family, Mom, Dad, Cindy, Shirley, Alex, also provided support and encouragement.

My friendship and interactions with the other active "hypertext versioning people," including David Durand, Fabio Vitali, David Hicks, Anja Haake, Jörg Haake, and Kasper Østerbye have broadened my understanding of these systems, and opened my eyes to many different perspectives. Andre van der Hoek's work on versioning and configuration management played a similar role. In part, you have motivated this dissertation by begging the question, "how can you explain all of our work?"

I would like to thank Ken Anderson for taking me "under his wing" as an apprentice on the Chimera project, my introduction to hypermedia. I'll never forget the long nights leading up to the Hypertext'93 demos, the thrill of our ECHT'94 acceptance, and the four year TOIS epic. Nor, the unforgettable "flying through your code with Chimera," video at STISS'95.

My understanding of the intricacies of containment was developed over a marathon series of teleconferences with the WebDAV Advanced Collections design team, Judy Slein, Geoff Clemm, Chuck Fay, Jim Davis, Jason Crawford, Kevin Wiggen, Jim Amsden, and Tyson Chihaya. Additionally, Dennis Hamilton provided insightful, detailed comments on an early draft of the containment model, and more generally has also helped me gain a broader understanding of document management. The DeltaV design team, Jim Amsden, Geoff Clemm, Chris Kaler, Tim Ellison, Henry Harbury, and Jim Doubek, through our debates concerning the design of the DeltaV protocol, helped me understand the design spaces associated with a configuration management system.

My understanding of document management was increased by my interactions with the WebDAV design team, Yaron Goland, Steven Carter, Del Jensen, Asad Faizi, and through interactions with Larry Masinter, and other members of the IETF WebDAV Working Group, acknowledged in RFC 2518. I am very appreciative of the contributions of long-time UCI WebDAV collaborators Joe Feise and Yuzo Kanomata for their work on the DAV Explorer, and DAV Posties; and Rohit Khare, for his help with political, promotional, and specification issues.

Throughout graduate school, I have benefited from a truly excellent cohort of other students and researchers at U.C. Irvine, including Ken Anderson, Roy Fielding, Rohit Khare, Beki Grinter, Greg Bolcer, Peyman Oreizy, Neno Medvidovic, Joe Feise, Yuzo Kanomata, Jeff Blevins, Peter Kammer, Lisa Covi, Leysia Palen, Debbie Dubrow, and Kari Nies. Thank you all for your friendship, support, discussions, and collaborations.

I would like to thank John Leggett, Nicole Yankelovich, Norm Meyrowitz, and Will Tracz for their emailed assistance tracking down the origins of the terms anchor and domain.

I'd like to thank my High School Physics teacher, Larry Smith, for providing computer access; he fanned flames of enthusiasm that last to this day. Finally, I'd like to thank Marty Greenstein, for encouraging me to pursue a Ph.D., and imagine it possible.

Curriculum Vitae

Emmet James Whitehead, Jr.

EDUCATION

Ph.D. Information and Computer Science, University of California, Irvine, December, 2000.

M.S. Information and Computer Science, University of California, Irvine, December, 1994.

B.S. Electrical Engineering, Rensselaer Polytechnic Institute, May 1989.

WORK EXPERIENCE

Chair, Web Distributed Authoring and Versioning (WebDAV) Working Group, Internet Engineering Task Force, March 1997-present. Founded and led this working group to enable Web-based remote collaborative authoring and versioning tools to be broadly interoperable. Assembled a broad coalition of participants from industry and academia, including Microsoft, Netscape, IBM, Novell, Xerox, Rational, Merant, Macromedia, and OTI. Developed the WebDAV Distributed Authoring Protocol, extensions to the Hypertext Transfer Protocol (HTTP) to support remote collaborative authoring, now supported by such industry-leading applications and servers as Office 2000, Go Live 5, Internet Explorer 5, Apache, Internet Information Services 5, Exchange 2000, DAV4J (IBM Alphaworks), and many others. Instrumental in the formation of the follow-on DASL (DAV Searching and Locating) and Delta-V (versioning and configuration management) working groups within the IETF.

Research Assistant, University of California, Irvine, 1993-2000. Performed research for DARPA research projects in the areas of remote collaborative authoring, open hypertext, hypertext versioning, software architecture, and software environments. Participated on several DARPA grant-writing teams.

Teaching Assistant, University of California, Irvine, 1992-1993. Assisted in the teaching of the undergraduate level Software Engineering course, and the Software Engineering Project course. Consistently earned excellent student reviews.

Engineer, Raytheon Equipment Division, 1989-1992. Wrote firmware in C and Ada for the German Civilian Air Traffic Control (ATC) System (DERD), and for a prototype Microwave Airplane Landing System (MLS). Adapted air traffic control software to create a marketing demonstration of a vessel traffic control system. Developed prototype touch panel displays for use with the DERD system. Project highlights include:

- Wrote firmware using the Spark safety-critical subset of Ada for an R3000-based single board computer to monitor the control electronics of a MLS in real-time. Wrote a C program to test a Phase Shifter Controller board. Debugged system hardware to the component level while assisting system integration.
- Wrote a multi-screen textual user interface for a real-time multiprocessor (four 68020 single board computers connected via a VME bus) simulator of nine radars, six direction finders, and a flight plan information system for the DERD ATC system.

PEER REVIEWED JOURNAL PUBLICATIONS

1. Kenneth M. Anderson, Richard N. Taylor, E. James Whitehead, Jr. “Chimera: Hypermedia for Heterogeneous Software Development Environments.” *ACM Transactions on Information Systems*, Vol. 18, No. 2, April, 2000.
2. Roy T. Fielding, E. James Whitehead, Jr., Kenneth M. Anderson, Gregory A. Bolcer, Peyman Oreizy, Richard N. Taylor “Web-Based Development of Complex Information Products.” *Communications of the ACM*, Vol. 41, No. 8, August, 1998, pages 84-92.

3. Kenneth M. Anderson, Richard N. Taylor, E. James Whitehead, Jr. “A Critique of the Open Hypermedia Protocol.” *Journal of Digital Information*, Vol. 1, No. 2, December, 1997.
4. Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow “A Component and Message-Based Architectural Style for GUI Software.” *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June, 1996, pages 390-406.

PEER REVIEWED CONFERENCE AND WORKSHOP PUBLICATIONS

Acceptance rates, where known, shown in parenthesis.

1. E. James Whitehead, Jr., Yaron Y. Goland, “WebDAV: A network protocol for remote collaborative authoring on the Web.” In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, Copenhagen, Denmark, September 12-16, 1999, pages 291-310. *(24%)*
2. E. James Whitehead, Jr. “Goals for a Configuration Management Network Protocol.” In *Proceedings of the Ninth Int'l Symposium on System Configuration Management (SCM-9)*, LNCS 1675, Toulouse, France, September 5-7, 1999, pages 186-203.
3. E. James Whitehead, Jr. “Control Choices and Network Effects in Hypertext Systems.” In *Proceedings of Hypertext'99, The 10th ACM Conference on Hypertext and Hypermedia*, Darmstadt, Germany, February 21-25, 1999, pages 75-82. *Engelbart best paper award nominee. (32%)*
4. E. James Whitehead, Jr. “Control Choices and Network Effects in Hypertext Systems.” In *Proceedings of the 4th Workshop on Open Hypermedia Systems*, held with Hypertext'98, Pittsburgh, PA, June 20-21, 1998.
5. E. James Whitehead, Jr. “An Architectural Model for Application Integration in Open Hypermedia Environments.” In *Proceedings of Hypertext'97, The Eighth ACM Conference on Hypertext*. Southampton, UK, April 6-11, 1997, pages 1-12.
6. Uffe Kock Wiil, E. James Whitehead, Jr. “Interoperability and Open Hypermedia Systems.” In *Proceedings of the 3rd Workshop on Open Hypermedia Systems*, held with Hypertext'97. Southampton, UK, April 6-7, 1997.
7. Nenad Medvidovic, Richard N. Taylor, E. James Whitehead, Jr. “Formal Modeling of Software Architectures at Multiple Levels of Abstraction.” In *Proceedings of the California Software Symposium 1996*. Los Angeles, CA, April, 1996, pages 16-27.
8. E. James Whitehead, Jr., “SCM and Hypertext Versioning: A Compelling Duo.” Position paper for *Sixth International Workshop on Software Configuration Management*, held with ICSE18. Berlin, Germany, March, 1996.
9. E. James Whitehead, Jr., Roy T. Fielding, and Kenneth M. Anderson, “Fusing WWW and Link Server Technology: One Approach.” In *Proceedings of the 2nd Workshop on Open Hypermedia Systems*, held with Hypertext'96. Washington, DC, March, 1996, pages 81-86.
10. Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins “A Component- and Message-Based Architectural Style for GUI Software.” In *Proceedings of the Seventeenth International Conference on Software Engineering*. Seattle, WA, April, 1995, pages 295-304. *(18%)*
11. E. James Whitehead, Jr., Jason E. Robbins, Nenad Medvidovic, Richard N. Taylor “Software Architecture: Foundation of a Software Component Marketplace.” In *Proceedings of the First International Workshop on Architectures for Software Systems*, held in cooperation with ICSE-17. Seattle, WA, April, 1995, pages 276-282.
12. Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead, Jr., “Chimera: Hypertext for Heterogeneous Software Environments.” In *Proceedings of the 1994 European Conference on Hypermedia Technology, ECHT'94*. Edinburgh, Scotland, September, 1994, pages 94-107.

13. E. James Whitehead, Jr., Kenneth M. Anderson, Richard N. Taylor, "A Proposal for Versioning Support for the Chimera System" In *Proceedings of the Workshop on Versioning in Hypertext Systems*, held with ECHT'94. Edinburgh, Scotland, September, 1994, pages 45-54.

INTERNET STANDARDS

1. Y. Goland, E. Whitehead, A. Faizi, S. Carter, D. Jensen, "HTTP Extensions for Distributed Authoring - WEBDAV." Microsoft, U.C. Irvine, Netscape, Novell, Internet Proposed Standard Request for Comments 2518. February, 1999.
2. E. Whitehead, M. Murata "XML Media Types." U.C. Irvine, Fuji Xerox Info. Systems, Internet Informational Request for Comments 2376. July, 1998.
3. Judith Slein, Fabio Vitali, E. James Whitehead, Jr., David Durand "Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web." Xerox Corporation, University of Bologna, U.C. Irvine, Boston University. Internet Informational Request for Comments 2291. February, 1998.

NON PEER REVIEWED PUBLICATIONS

1. Jim Whitehead, "The Future of Distributed Software Development on the Internet." *Web Techniques*, Vol. 4, No. 10, October, 1999, pages 57-63.
2. E. James Whitehead, Jr. "Collaborative Software Engineering on the Web: Introducing WebDAV." *Software Tech News*, Vol. 3, No. 1, 1999, pages 5-9.
3. E. James Whitehead, Jr. "Collaborative Authoring on the Web: Introducing WebDAV." *Bulletin of the American Society for Information Science*, Vol. 25, No. 1, October/November, 1998, pages 25-29.
4. E. James Whitehead, Jr., Meredith Wiggins, "WebDAV: IETF Standard for Collaborative Authoring on the Web." *IEEE Internet Computing*, Vol. 2, No. 5, September/October, 1998, pages 34-40.
5. E. James Whitehead, Jr., "Lessons from WebDAV for the Next Generation Web Infrastructure." In *Towards a New Generation of HTTP, A workshop on global hypermedia infrastructure*, held with 7th Int'l World Wide Web Conference, Brisbane, Queensland, Australia, April 14, 1998.
6. E. James Whitehead, Jr. "World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction." *StandardView*, Vol. 5, No. 1., March 1997, pages 3-8.
7. Gail Kaiser, Jim Whitehead, "Collaborative Work: Distributed Authoring and Versioning." Column in *IEEE Internet Computing*, Vol. 1, No. 2, March/April, 1997, pages 76-77.
8. Jeffrey J. Blevins, E. James Whitehead, Jr., Harry E. Yessayan "Report on The Software Environments Technical Research Review." In *Proc. 4th Irvine Software Symposium*. Irvine, CA, April, 1994, pages 89-96.

WORKSHOPS, TUTORIALS, PANELS, AND PRESENTATIONS

Organizing Committee, "TWIST 2000, The Workshop on Internet Scale Software Technologies: Organizational and Technical Issues in the Tension Between Centralized and Decentralized Applications on the Internet", July 13-14, 2000, Irvine, CA.

Organizing Committee, "TWIST'99, The Workshop on Internet Scale Software Technologies: Internet-scale Namespaces", August 19-20, 1999, Irvine, CA.

Workshop Co-Chair, "WISEN: Workshop on Internet Scale Event Notification", July 13-14, 1998, Irvine, CA.

Workshop Co-Chair, "Towards a New Generation of HTTP, A workshop on global hypermedia infrastructure," held in conjunction with the 7th Int'l World Wide Web Conference, Brisbane, Queensland, Australia, April 14, 1998.

Taught the tutorial, "The Web as a Writeable, Collaborative Medium: An Introduction to the IETF WebDAV Standard" at Hypertext'99, the 10th ACM Conference on Hypertext and Hypermedia, February 22, 1999, Darmstadt, Germany.

With Rohit Khare, taught the tutorial, “XML and WebDAV: Emerging Web standards and their impact on software engineering” at SIGSOFT'98, The Sixth International Symposium on the Foundations of Software Engineering (FSE-6), November 5, 1998, Lake Buena Vista, FL.

With Rohit Khare, taught the mini-tutorial, “Web Automation and Collaboration: XML, WebDAV, and ISEN” at the 1998 California Software Symposium, October 23, 1998, Irvine, CA.

With Josh Cohen, taught the tutorial, “WebDAV, the Web Document Authoring and Versioning Effort” at the Hypertext'98 Conference, June 21, 1998, Pittsburgh, PA.

Participated on the panel, “Collaborative Authoring and Electronic Document Management,” held at the American Society for Information Science (ASIS) 1998 Mid-Year Conference, Orlando, Florida, May 19, 1998.

Participated on the panel, “Missing the 404: Link Integrity on the World Wide Web,” held at the 7th International World Wide Web Conference, Brisbane, Queensland, Australia, April, 1998. Panel abstract: *Proc. 7th Int’l World Wide Web Conference*, Computer Networks and ISDN Systems, Vol. 30, Nos. 1-7, April 1998, p. 761-762.

Participated on the panel, “Things Change, Deal With It! Versioning, Cooperative Editing, and Hypertext,” held at Hypertext'96, Washington, DC, March, 1996. Panel abstract: *Proc. Hypertext’96*, p. 259.

Presented “Web-Based Remote Collaborative Software Engineering with WebDAV” at the 1999 Software Technology Conference (STC'99), May 6, 1999, Salt Lake City, Utah.

With Martin Cagan, presented “World Wide Web Content Management: Emerging Problems, Emerging Solutions,” at *Wired: The Internet and the World Wide Web*, a symposium hosted by the Irvine Research Unit in Software, May, 1996.

PROFESSIONAL SOCIETIES

Association for Computing Machinery (ACM), Institute of Electrical and Electronics Engineers (IEEE), Internet Society (ISOC), American Society for Information Science (ASIS)

SIGNIFICANT VOLUNTEER ACTIVITIES

1991 Instructor. Created and taught an 8 week course on C language programming to the Amiga Users Group of the Boston Computer Society.

1993-1996 Host, Cyberspace Report Radio Show. Conducted over 80 interviews on social issues of computing for this 30 minute weekly radio show broadcast on KUCI. Notable guests include Ted Nelson, inventor of hypertext, Richard Stallman, founder of the Free Software Foundation, Emmanuel Goldstein, editor of 2600 magazine, and Roy Fielding, an architect of the HyperText Transfer Protocol. The Cyberspace Report web page was one of the earliest radio shows to exploit the WWW, and the collection of interview tapes comprise a significant oral history archive from the early years of the Web.

1993-1994 Vice President of Financial Affairs, Associated Graduate Students of the University of California, Irvine. Responsible for opening and operating a bar in the U.C. Irvine Student Center with $92K in sales. Administered a $75K budget. Served on the Student Fee Advisory Committee which made recommendations on the U.C. Irvine budget to the Chancellor.

1995-2000 Treasurer, Rachel Carson Organic Garden Cooperative. Key participant in the design, procurement of funding, purchasing of materials, and construction of the garden. The project required $7.5K of funding, and includes 26 raised garden beds, and running water to the garden site. Administered cooperative finances.

Abstract of the Dissertation

An Analysis of the Hypertext Versioning Domain

By

Emmet James Whitehead, Jr.

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2000

Professor Richard N. Taylor, Chair

Hypertext captures the implicit and explicit relationships between intellectual works, storing them as data items within the computer, thus allowing them to be navigated, analyzed, and visualized. The evolution of information artifacts such as software development projects, large document collections, and collections of laws and regulations is characterized both by change to the works and their relationships, and the desire to record this change over time. Hypertext versioning is concerned with storing, retrieving, and navigating prior states of a hypertext, and with allowing groups of collaborating authors to develop new states over time.

Several systems provide hypertext versioning services; this dissertation provides a domain model of these systems, comprised of domain terminology, a taxonomy, reference requirements, a data modeling model, and design spaces associated with the requirements. This work offers several significant contributions. It provides a systematic organization of the preponderance of information concerning hypertext versioning systems, including the first taxonomy of such systems, and a comprehensive collection of their requirements. A detailed model of containment is provided; its use highlights that containment is inherent in hypertext systems, and a full understanding of hypertext versioning system data models requires an understanding of their containment relationships. The containment model allows the similarities and differences in hypertext versioning systems to be examined in a consistent manner.

The design space for persistently recording revision histories employs a three-layer model that separates of the abstract notion of revision history, shared by all state-based approaches, from the high-level overview of each versioning approach, which is in turn distinct from its specific concrete representation. The design space for link versioning is shown to be an application of the three-layer model for versioning works. Building on the containment model, and the design spaces for versioning works and links, the structure design space concisely describes a range of techniques for recording the history of hypertext structures. Parameters of the structure design space include the abstractions contained within the structure container, the versioning design space choice for each versioned abstraction, the containment choice for each container/containee pair, and the location of any revision selection rules.

Chapter 1

Introduction

1.1 Hypertext, and Hypertext Versioning

In the absence of hypertext, our data is tightly packaged. Documents, spreadsheets, presentations, source code, and databases, all exist wrapped up in their snug, independent files, each datum an island. Separating information *content* from its physical or logical *container*, it is clear that contents are anything but isolated. Immersed in a bramble of associations, the contents make explicit references via annotation, footnote and citation, and implicit references via similarity (or difference) in content, style, location, history, or goal. Standards, institutions, and community norms impact form and structure in almost invisible ways, as with academic citation style, and publication-specific length and formatting rules. Michael Foucault captures this distinction of unitary package and boundary-free content when he writes:

> The frontiers of a book are never clear-cut: beyond the title, the first lines, and the last full stop, beyond its internal configuration and its autonomous form, it is caught up in a system of references to other books, other texts, other sentences: it is a node within a network. … The book is not simply the object that one holds in one's hands; and it cannot remain within the little parallelepiped that contains it: its unity is variable and relative. [71], p. 23

The notion of a singular document relating to other documents is ancient. Religious texts across cultures have included and attracted annotations, texts providing discussion and commentary on holy scriptures. Roman jurists referenced other legal decisions and precedents precisely, though other Roman authors only rarely referenced the authors, and less frequently the titles, of works on which they based their texts. More typically they quoted passages from memory, often introducing a slight change to show this. Later, in the 12th century schools that evolved into universities, medieval scholars developed precise and standardized reference forms [82], p. 26-30. The modern footnote, distinct from annotation and citation, can be traced back to at least 1696, and the publication of “Historical and Critical Dictionary” by Pierre Bayle, an ambitious volume that sought to document all errors and omissions in existing historical reference books [82], p. 192-197. By the middle of the 20th century, the practice of documents containing footnotes, as well as annotating and citing other documents, was common in most branches of academia.

In 1945, Vannevar Bush made the critical leap that inter-document relationships could be *mechanized*, stored in an optical, electrical, and mechanical device named the Memex, thereby allowing a reader to quickly follow document associations, nimbly hopping from one document to the next near the speed of thought [25]. Though the proposed Memex machine employed microfilm, it was obvious to future researchers that the computer’s ability to flexibly manipulate text made it the ideal medium for document interlinking, creating texts that are more than just text: *hyper*texts. A computer can store large volumes of information compactly, provide searching over the information, and, when combined with a network, can import and export this information. When following a relationship, it can highlight the related text, often by amending the original document to underline regions that are part of relationships. Relationships in computer storage can themselves be analyzed, displayed separate to their documents, and link trails can be exchanged among interested users. The computer makes it easy to change documents and the network of relationships between them, simultaneously a blessing and a curse.

Hypertext captures the implicit and explicit relationships between individual files, making them real data objects that can be acted upon by the computer. But, this act of capturing the relationships breaks down the packaging of information into individual files. Since files are dynamic, changing, a tension develops between two views: the one presented by current tools where users have the impression of acting on files in isolation, and the hypertext view of files participating in rich networks of relationships where each modification can cause changes that propagate into the network.

Consider software engineering. A large software project consists of many thousands of files, comprising requirements and design documents, source code, test cases, build files, bug reports, memos, email, and Web pages. There are many relationships between these files, such as a source file that satisfies a requirement stated in another document, or a test case that examines whether the code does indeed meet that requirement. In fact, software project files have an enormous number of relationships between them, and hence the project is really more a *complex information artifact* than a mere collection of files [70].

Chimera [7] and DHM [84] are two examples of hypertext systems whose goal is to capture the relationships between software project files. Once these relationships are in the hypertext system, they allow for rapid navigation to related files, as well as visualization and analysis of the relationship network. The act of instantiating the relationships makes concrete the effect that changing a single software file can have on its network of relationships, since modifying a file can create new relationships, and can alter or destroy existing ones.

Software engineering is a domain where best common practice involves maintaining previous states of the project. The discipline of software configuration management has developed to address the difficult issues of how best to record and compose these previous states, allow teams of developers to work on them without clobbering each other's work, guide and audit the software development process, and produce statistics based on this historical data [39]. However, since software development projects are currently divided into files, configuration management systems typically provide their features for files. Once hypertext functionality is added to a software development project, there is an immediate tension between file-oriented configuration management, and network-oriented hypertext functionality.

Think of a collection of project files whose previous states have been saved. In the absence of hypertext, there are still relationships between these files, but they are not captured in a hypertext system. By picking and choosing individual revisions of each file, it is possible to create arbitrary compositions of file revisions, even though some compositions may have relationships that are inconsistent. However, once the relationships are made into an explicit hypertext, this is no longer the case. Since the relationships change over time, they too must have their previous states recorded. The hypertext links furthermore make it obvious when picking a single file revision causes a hypertext relationship to become inconsistent.

In hypertext parlance, a *node,* or *object*, is a chunk of data that can be stored as a file, a database record, or even, as is the case with Web pages in embedded devices, a sequence of read-only memory. Dynamic objects can be arbitrary computational processes, a common occurrence on the Web. Relationships between objects are called hypertext *links*. Though the detailed specifics of links vary across systems, in general a hypertext link associates two or more objects, providing a means to navigate quickly between them. Links can be either to entire objects, or to a specific region, called an *anchor*, within an object. At their most abstract, anchors and links can both be arbitrary computational processes, one example being an automatic link between any word in a document, and its entry in a dictionary. This document concentrates its discussion primarily on non-dynamic objects, links, and anchors, reflecting both the fact that the preponderance of existing hypertext versioning systems do not provide versioning operations to handle such dynamism (an exception being [100]), and that the problem is inherently complex.

A collection of objects, links, and anchors comprises a *hypertext document*, more concisely known as a *hypertext*. *Hypertext versioning* is concerned with storing, retrieving, and navigating prior states of a hypertext, and with allowing new states of the hypertext to be created by groups of collaborating authors.

Hypertext versioning capability has a pervasive impact on a hypertext system. It affects the storage of objects, anchors, and links, the way people collaborate using the system, how navigation occurs, the naming of objects, and can reduce system performance. Hypertext versioning is an expensive proposition, since adding this functionality directly increases the complexity of a system for both implementers and users. Given its scope of impact and complexity cost, it is reasonable to ask whether the functionality is worth the trouble. What, exactly, can hypertext versioning be used for? High-value use cases include:

- **Software engineering.** As mentioned above, capturing the evolution of development files, as well as the relationships between them provides significant advantages for software development. However, due to the prevalent use of versioning and configuration management in software development environments, in order to provide hypertext support, the hypertext structure must also be versioned. The fact that existing hypertext systems for software development do not version links is a significant factor preventing their wider use in this domain.
- **Document management.** Documents too have a wide range of relationships between them, and can benefit from using hypertext to make them explicit so they can be analyzed, visualized, and navigated. Document management systems today provide the ability to store important document states, thus supporting collaboration and backtrack. Hence, just as for software engineering, hypertext support for document management requires versioning of structure in concert with the versioning of documents.
- **Legal.** Laws, regulations, and tax codes, are an important set of complex information artifacts, chock full of interrelationships. It is important to store and retrieve previous document revisions

because in legal systems that prevent ex-post-facto laws, the version of a law that affects a case is the one in effect at the time of an infraction. This is especially relevant for tax codes, which change frequently. Hypertext support can make it easy to navigate to related laws, precedents, regulations, and codes. In this domain as well, adding hypertext support requires hypertext versioning capability.

- **Archival**: Since the Web is an important cultural artifact, the development of the Web should be recorded as it evolves. Right now, though the Internet Archive group is archiving the Web [32,105], only preliminary research has been performed on creating a "way back" machine that allows a user to dial in a specific time, and then be able to navigate around as if they were interacting with the Web as of that day [66]. Archiving fills legal needs too. Regulated firms, such as insurance or brokerages that advertise or sell their products via the Web, have an obligation to archive their web sites so they can recover previous states for use in lawsuits.
- **Auditing**. During an audit, a team of auditors gathers and condenses information about a company to develop an independent opinion concerning the accuracy of its financial statements. Auditors create linkages between documents to create a network of content substantiation, in the process creating files called audit working papers [49]. Due to the collaborative nature of the task, the need to freeze a state of the company's documents for analysis, along with the emergent understanding of the financial statements made by the company and the change this implies to inter-document linkages, the audit working papers, and the final audit report, hypertext versioning is necessary for the introduction of hypertext into financial audits.
- **Reference permanence.** One solution to the common Web problem of dangling links ("404 Not Found") is to ensure the linked-to information is always intact, by recording its prior states. In this way, link endpoints will always be present.

In addition to its utility in these use cases, hypertext versioning research offers much to the long-term goals of building a better Web, and the convergence of collaborative repositories.

At present, linking on the Web consists only of tags embedded in HTML. This is changing. The Xlink proposal [47] provides for linking between XML [24] documents, and Xlinks can be stored external to the documents they reference. The ability of open hypertext systems to link between arbitrary data types, including legacy formats that support neither embedded HTML-style links, nor Xlinks, is another driver pushing the Web towards providing links that are stored separately from the data. Of course, these links will change over time, and will require version control. How best should this functionality be provided in the context of the Web?

There is a growing convergence of functionality across multiple types of repositories that support the development of complex information artifacts, such as document management systems, configuration management systems, and Web content management systems. This convergence is most evident in the Web, with the proven track record of HTTP [68] and WebDAV [79] to map to a wide range of repositories, and with Delta-V [199] extending this to configuration management repositories as well. These repositories today do not provide support for first-class linking, and it is an open question how they might provide the hypertext versioning combination of first class linking and version support in the future. Since there is no current research consensus on how best to provide this functionality, it is difficult to make a case for its standardization, since such standardization is probably premature.

Today, hypertext technology has matured to the point where it is used widely to capture and express relationships in read-only information spaces. However, the benefits associated with adding hypertext capability to the collections of documents and artifacts in software engineering, document management, legal, and auditing highlight the utility of expanding hypertext use beyond read-only information spaces. Adding hypertext capability to these writeable information spaces necessitates engaging compatibility issues, such as interacting with legacy information stores and application programs. These issues have been a mainstay of the open hypertext literature, addressed in systems such as Chimera [7], and Sun's Link Service [147]. However the interactions between hypertext, versioned data, and versioned containment structures, is much less understood. While there has been a slow but steady stream of research on hypertext versioning, the knowledge generated has been specific to the data model and system on which the research was performed, and is difficult to apply to different systems or problem areas. But, given the importance of versioning to domains such as software engineering and document management, the application of hypertext to complex information artifacts is limited by the absence of systematically organized knowledge concerning hypertext versioning.

Having identified the need for systematic hypertext versioning knowledge, how should this knowledge be organized? Since the goal of this knowledge organization task is the reuse of hypertext versioning concepts and techniques in multiple support environments for complex information artifacts, it is not too surprising that the answer should come from the Software Reuse community, in the form of domain analysis. Domain analysis provides a ready-made structure for organizing a large body of knowledge about a specific problem area, as well as a rich vein of articles and books providing tutorials, analysis, and examples of this technique. This dissertation performs an analysis of the domain of hypertext versioning, thereby building on the strong foundation of domain analysis.

1.2 Domain Analysis

Creating software out of reusable components, rather than modules tailor-made for a specific application, is a goal as old as Software Engineering itself [127]. Component-based software development yields many benefits, including decreased system development time and cost (although the initial development cost for reusable components is higher), reduced per-instance component cost due to the amortization of component development costs over multiple users, increased component robustness due to greater maintenance resources supported by multiple users of each component, and a wide range of general-purpose and domain specific components. Reusable components fall broadly into two categories, horizontal, and vertical. Horizontally reusable components are those that can be used across a wide range of application programs, with examples including data structures, common algorithms, and user interface toolkits. Vertically reusable components are tailored to a specific problem area, and can be reused in applications within that domain [102]. The Common Ada Missile Package (CAMP) ([26] as referenced in [156]) and the Avionics Domain Application Generation Environment (ADAGE) project [16] are two examples of vertical component reuse.

Domain models are typically created during a process of domain analysis for the purpose of fostering vertical component reuse. One insight motivating the creation of domain models is that code alone is insufficient to generate reuse, since the analysis and modeling of the problem space is only partially reflected in the code itself. Furthermore, it is difficult to reconstruct those aspects of the problem space that are reflected in the code just by examining it [136]. Reuse is also hindered when individual components have been constructed independently, without coordination, and cannot be used together due to architectural mismatch, such as differences in control assumptions, processing requirements, memory usage, etc. Domain models increase opportunities for reuse because they allow components to be designed for a specific domain, to fit within a specific domain architecture. In domains that can be formally modeled, it is also possible to use program generation technology to automatically create domain components [136,16].

Domain analysis is fraught with specialized terminology, and some common terms are defined below:

Domain. A coherent problem area [83].

Model. An abstract representation of some process, phenomenon, or entity [131].

Domain model. A model of a domain, consisting of terminology used in the domain, a description and parameterization of the domain problem area, as well as a characterization of elements and relationships in the domain [184].

Reference requirements. A collection of goals concerning the functionality and behavior of applications in a domain.

Domain terminology. Terms, and their definition, for significant concepts, abstractions and processes in the domain.

In its original meaning, a domain is all of the lands controlled by a lord, that is, the realm of the lord [158]. When applied to ideas, domain is an analogy, where ideas are synonymous with land, and the ruler is the particular idea or abstraction. In this case, it is the idea that defines the portion of land under its rule, that is, the idea creates the space it rules. When applied to Software Engineering, a *domain* represents all of the possible aspects of a particular problem or task area. Following the domain-as-idea analogy, an abstract problem area defines a domain, creating a space of problem aspects, captured as a set of *reference requirements*.

Automatic generation of components via program transformation motivated the original work on domain analysis and modeling. Jim Neighbors' dissertation, "Software Construction Using Components," [136] is generally credited with originating the concept of domain analysis, and domain modeling (for example, see [156], p. 48). However, in the related work chapter of his dissertation, Neighbors quotes several motivating sources, including Julian Feldman, Tim Standish, and Peter Freeman at U.C. Irvine, and in this quote attributes to Bob Balzer the notion of creating a model of a problem domain:

> A model of the problem domain must be built and it must characterize the relevant relationships between entities in the problem and the action in that domain. [14], quoted in [136]

However, Balzer himself does not take credit for the term, pointing instead to its use in Artificial Intelligence to discuss knowledge specific to a particular problem area—and perhaps amenable to capture in an expert system—as distinct from general knowledge. Even though the notion of a domain was in use within the Artificial Intelligence community, and the idea of domain modeling was "in the air" at U.C. Irvine and in the program transformation community in the 1970's, Neighbors' dissertation is clearly a watershed in terms of collecting the notion of domain modeling and analysis together into a coherent, organized manner. Furthermore, the idea of domain analysis continued to be active within Peter Freeman's group at U.C. Irvine, showing up in the work of his students, such as Guillermo Arango in his 1988 dissertation, "Domain Engineering for Software Reuse" [10] and the later work by Rubén Prieto-Díaz on multi-faceted domain analysis and modeling [155,156].

A domain should have a crisply defined boundary, thus ensuring that a well-defined domain can easily be differentiated from other domains. A crisply defined problem area is one that has a high degree of relatedness and similarity to aspects of the problem. Typically a domain has a common *domain terminology* associated with it, used to describe typical abstractions and processes that are associated with the domain. Once the problem aspects have been captured as reference requirements, they can be used to develop a class of similar application systems that meet potentially subsetted, parameterized, or modified reference requirements for the specific application. This class of similar systems that address a particular domain is very similar to the Parnas notion of program families [146], and the more recent notion of product lines [83].

Typically, some class of systems has been developed and fielded before there is recognition that the systems are similar, and the problem area they address constitutes a specific domain. In this case, it is possible perform a process called *domain analysis* on the existing systems to determine their requirements, data models, architectures, along with the interrelationships and similarities among these items [99,184].

The first output of domain analysis is a characterization of the domain itself, in the form of a *domain taxonomy*, a classification of applications and their software components within the domain, *domain terminology*, terms and their definitions for important concepts, abstractions, and processes in the domain, and *reference requirements*, a collection of goals concerning the functionality and behavior of applications in a domain. Requirements for a specific application are developed by parameterization, modification, and selection of the reference requirements.

Domain analysis also creates a *domain model*, which represents the elements and their relationships in the domain. The domain model is an abstract representation of the domain, and is distinct from a *reference architecture*, which is a software architecture for a family of systems within the domain. While a domain model contains the primary abstractions employed in the domain, relationships between the abstractions (such as containment, inheritance, etc.), and the design space of tradeoffs that can be employed to satisfy the domain's reference requirements, a reference architecture assigns the abstractions and functions to specific modules, defines interfaces for these modules, specifies interrelations among the modules, and resolves or parameterizes many of the design tradeoffs [131,184]. Though the reference architecture does not necessarily describe a concrete implementation, it still provides a framework in which reusable components can be developed for the domain, and, importantly, it can form the basis for automatic generation of components [16,17].

An important goal of systems work in Computer Science is the thorough exploration of a particular domain, with individual systems exploring specific aspects of the domain, ideally by making particular design choices that are different in some respect from existing systems. Once enough systems have been developed, a process of condensing the knowledge of the domain into a coherent domain model takes place. This process of distillation will suggest additional avenues of research, and may itself require multiple iterations before the domain model is considered complete. This work takes on more of the aspect of a survey, though considerable creativity is required to tease out a coherent domain model from individual instances of the domain. A good domain model includes multiple frameworks, each of which is a significant original intellectual activity.

Domain model in hand, several other avenues open up. The definition of the domain can be relaxed, allowing other, similar systems into the domain. One example is relaxing hypertext versioning to include configuration management, document management, and engineering database systems. This may take the form of combining multiple individual domain models from similar domains, or by expanding one existing model to encompass a neighboring domain.

Domain analysis makes it possible to tease out, or abstract, aspects of the domain model that can apply across multiple domains, or across all domains. Domain-independent knowledge is very powerful, since it can be applied to many possible domains. Design patterns are one form that domain-independent knowledge can take [76]. There are others. Thus, one of the major benefits of domain analysis is the ability to take knowledge that is currently embedded in the domain, and make it *portable*, applicable across domains.

1.3 Domain Analysis of Hypertext Versioning

We have noted that the goal of this dissertation is to provide a domain analysis of hypertext versioning, and that domains ideally have a well defined boundary. How, then, is the domain of hypertext versioning defined?

First and foremost, the hypertext versioning domain includes those systems that provide hypertext navigation of either versioned or unversioned links among versioned objects. By insisting on hypertext navigation features, systems that only provide some form of relationship, such as relational database systems, or software development environments, are excluded. Despite supporting hypertext navigation, the majority of systems in the domain emphasize characteristics of their data models (e.g. [141], [100]), to the extent that there is a significant lack of research on how best to visualize and create user interfaces for navigation through versioned hypertext structures. However, cleaving a distinction between hypertext links and general relationship features is fairly arbitrary and artificial, especially since we will later model links as a kind of container object, a view that emphasizes similarities between links and relationships. Restricting the domain to navigable hypertext links can be viewed as a way to limit the scope of this inquiry, while the similarity of links and relationships clearly point to a future expansion of this work to all object management systems employing relationships.

In many respects, the domain of hypertext versioning is a post-hoc attempt to provide an organizing principle for the many systems deemed to be hypertext versioning by the research community. As a result, though defining the domain as containing just systems that provide versioned or unversioned links among versioned objects encompasses most of the hypertext versioning research, there are some exceptions to this rule. Since an explicit design goal of Palimpsest [56] and VTML [194] is to provide a data structure supportive of collaborative work and reference permanence, they are included even though they provide no explicit link traversal capabilities. Another exception is the PIE system [80], which is included due to the significant influence its hypertext-like data model has had on several hypertext versioning systems, despite the fact it has no hypertext navigation, and makes no claim to be a hypertext system. These exceptions highlight that hypertext versioning is a domain at the crossroads of the hypertext, configuration management, document management, software development environments, and computer supported cooperative work research communities.

The results of this domain analysis are intended to provide a systematic organization of the domain of hypertext versioning, affording reuse of hypertext versioning concepts in a wide range of collaborative systems, thereby removing a major impediment to the application of hypertext to the complex information artifacts generated in software development, document management, audits, legal proceedings, and academic annotation and cross-referencing. Thus, a primary goal is *reuse of the products of the domain analysis*, such as the concepts, terminology, taxonomy, requirements, data modeling mechanism, design spaces, etc., and *not* code reuse. Domain analysis provides a strong framework for examining an entire family of systems, and a useful set of outputs from this analysis.

An idealized process for reusing the hypertext versioning domain model takes as its starting point an existing repository system, such as a hypertext, configuration management, or document management system, and augments it with hypertext versioning capability. The engineers of this system begin by evaluating the reference requirements (given in Chapter 7) one by one, selecting those requirements that satisfy the hypertext versioning needs of the system, rejecting those that are non-relevant. The selected requirements are next be evaluated to determine their interactions, and this analysis can cause requirements to be added, removed, or amended.

After requirements are selected, a containment data model is created of the system (using the data modeling model described in Chapter 6) *before* hypertext versioning is added. Using this model as a baseline, the design space associated with each reference requirement (described in Chapter 8) is be evaluated based on its technical characteristics, and its interactions with other design choices. Based on this analysis, a point in the design space would be chosen, and the containment data model updated with any new abstractions and containment relationships. Alternately, it might be found that the tradeoffs associated with meeting a requirement are too severe, and hence the requirement is dropped. Unfortunately, in some cases, there is insufficient knowledge about how to meet a given requirement, and only general guidance can be offered; such is the case, for example, with versioned hypertext visualization (Section 8.10), still an open research problem. Once all requirements have been satisfied, the system can be evaluated for emergent properties of the combination and interaction of multiple design spaces. This may lead to reevaluation of requirements, and design choices, and another iteration through the process. At the end of the idealized process, the system designer will have the set of desired requirements, the set of design choices selected, a containment data model of the system before and after hypertext versioning capability was added, and an understanding of the tradeoffs involved in the requirements and design choices he made.

This analysis of the domain of hypertext versioning systems yields a characterization of the domain, and a domain model. The domain is characterized by:

- **Domain terminology:** Terms, and their meaning, that are used to describe fundamental abstractions in hypertext versioning systems. These abstractions are the fundamental building blocks for the rest of the domain model: all other aspect of the domain model either list constraints, or describe relationships between these abstractions. The domain terminology is presented in Chapter 2.
- **Taxonomy of systems:** Systems considered to belong to the hypertext versioning domain are analyzed for similarities. Based on this analysis, the domain is divided into five classes of systems, listed in Chapter 3.
- **Reference requirements:** A complete set of requirements for the versioning aspects of hypertext versioning systems, based on an analysis of systems in the hypertext versioning domain. The reference requirements are presented in Chapter 7.

The hypertext versioning domain model is composed of:

- **Data modeling model:** A data model of the key entities and their relationships within hypertext versioning systems. An extended entity-relationship model [29], an important member of the class of semantic data models [148,103], is used to represent the entities, and their relationships. An important, and frequently occurring relationship among entities is containment, and this relationship is described in depth in Chapter 4. Referential containment structures point to the data they contain, and do so using names or addresses provided by the system. An overview of address and name spaces in hypertext versioning systems is provided in Chapter 5. Containment model in hand, the data modeling model, and examples of systems represented using this model, are given in Chapter 6.

- **Design spaces associated with domain requirements:** For each of the domain requirements given in Chapter 7, Chapter 8 describes the design choices available to satisfy the requirement. Though the design spaces are presented in the same order as the requirements, there are several important design spaces worthy of individual note:
    - *Data versioning* describes the design space of persistent storage for object revisions (Section 8.2).
    - *Link and structure versioning* describes the design space of persistently recording link revisions, and link structure revisions (Section 8.5).
    - *Variant support* describes how alternate forms of an object are persistently recorded (Section 8.6).
    - *Collaboration support* describes how cooperating workers can ensure that they do not overwrite each other’s work, both for individual objects, and for collections of objects (Section 8.7).

The description of design spaces associated with each domain requirement takes this work beyond that of a survey of hypertext versioning systems. In a typical survey, the emphasis is on describing the behavior of existing systems. In contrast, the design spaces describe characteristics and constraints on systems that *could* be built in the domain, and hence has a constructive, rather than a descriptive emphasis.

Several additional chapters round out this work.

- **Validation.** Despite being based on existing hypertext versioning systems, to ensure that the domain model contains sufficient modeling power to describe existing hypertext versioning systems, Chapter 9 applies the domain model to the Chimera hypertext versioning proposal [200], and the DeltaV protocol for Web versioning and configuration management [199]. In this way the domain analysis is validated.

- **Related work.** Chapter 10 provides a description of related domain modeling efforts, as well as surveys in hypertext versioning, configuration management, and engineering databases.
- **Future work.** The systematic exploration of the hypertext versioning domain has uncovered many areas within hypertext versioning that require additional research. Furthermore, the domain analysis itself can be expanded to other domains. These future directions are discussed in Chapter 11.
- **Contributions and Conclusions.** A summary of the original contributions made in this document are presented in Chapter 12.

Chapter 2

Domain Terminology

This chapter provides the name, definition, and origin for key abstractions in the hypertext versioning domain, divided into hypertext terms, and versioning terms. However, before definitions for hypertext and versioning terms can be given, some general terms need to be defined.

2.1 General Terms

Abstraction. An idea or concept having a crisply defined boundary.

Entity. A signifier for an abstraction, used in modeling.

Object. A single or aggregate data item that represents an entity or abstraction.

Data Item. A single structured or unstructured conglomerate of data.

Hypertext versioning systems are distinguished by their different models. Whether it be document models, version history models, or workspace models, hypertext versioning systems all embody a collection of models. But, what are these models built from?

All models begin in the mind, as discrete thoughts. We use the term *abstraction* to represent thoughts that have been sufficiently refined as to have crisply defined boundaries. The notion of abstraction is ancient; what we call abstraction is nearly identical to the Platonic notion of Form [153]. Since abstractions exist only in the mind, but yet models need to exist outside the mind to be communicated and manipulated, there is a need to create signifiers, or handles, for abstractions. We term these *entities*. Symbolic representations of entities appear, for example, in entity-relationship diagrams [29].

The representation of abstractions within the computer concerns their mapping into data items. Individual *data items* are groups of data such as an unstructured sequence of bytes, or some data structure, like a queue or a tree. An *object* is either a single data item, such as a Unix file, which is a sequence of bytes, or it can be an aggregation of data items. Objects can have a single main data item, representing the primary state of the object, along with a series of data items holding metadata. Alternately, an object can be a collection of data items, with no distinction between primary, and metadata items. The data organization of an object represents a particular entity, or abstraction.

2.2 Hypertext

> *Hypertext.* A set of intellectual works and their inter- and intra-work associations, represented as links, in combination with a user interface for viewing instances of these works and quickly navigating from instance to instance across links.

Coined in the mid-1960's by Ted Nelson, the term *hypertext* conjoins *hyper* and *text*. Hyper, used as a prefix, derives from the Greek hypér, originally meaning over, or above, but whose meaning typically implies excess or exaggeration. A synonymous prefix is *super* [158]. There is also the independent meaning of *hyper* used as a noun to mean, "a person who promotes or publicizes events, people, etc., esp. one who uses flamboyant or questionable methods; promoter; publicist" [158]. Text has the original meaning of words woven together [158], and so combined with hyper, hypertext implies both a super text, a text that, due to interlinking, is greater than the original texts, and a super weaving of words, creating new texts from old.

Hypertext simultaneously means the *concept* of interlinking texts, an *instance* of this concept as a software system and collection of interlinked texts, a *data model* that supports text interlinkage, and the *academic discipline* whose concern is hypertext. The alternate term *web*, first used by Intermedia (e.g., as found in Figure 11 of [33]) and popularized by the World Wide Web, also represents an instance of a collection of interlinked texts. Within this dissertation, hypertext will be used with one of the first three meanings, concept, instance, and data model, though this document as a whole exists as a contribution to the academic discipline of hypertext.

Given the struggle Nelson encountered in disseminating the idea of hypertext, it is possible to view the word hypertext acting as a promoter and publicist, carrier of the linked text meme. Nelson writes:

> I coined the term "hypertext" over twenty years ago, and in the ensuing decades have given many speeches and written numerous articles preaching the hypertext revolution: telling people hypertext would be the wave of the future, the next stage of civilization, the next stage of literature and a clarifying force in education and the technical fields, as well as art and culture. [137], p. 0/2

In fact, the flamboyant promotion of hypertext was such an integral part of the initial culture of the hypertext community that by 1987 Jeff Raskin's paper at the Hypertext'87 conference is titled, "The Hype in Hypertext: A Critique" [159] where he claims Nelson, "writes with the messianic verve characteristic of visionaries," and in 1989 Norm Meyrowitz's Hypertext'89 conference keynote is titled, "Hypertext—Does It Reduce Cholesterol, Too?" [129]. Clearly the notion of the interlinked text as super text is inseparable from hypertext as hyped concept.

It is a regular staple of hypertext papers to note that the term *hypermedia* also exists, and refers to interlinked non-textual objects, such as pictures, movies, etc. However, by defining hypertext in terms of interlinked objects that can be text, picture, movie, or any other type of content, the term hypertext encompasses the meaning of hypermedia. Given the choice of two equally expressive terms, the research community typically prefers hypertext to hypermedia, as do we.

The opening definition in this section stresses that hypertext is more than just the capturing of associations within the computer to create a network of works. Though central to hypertext, it is the combination of recording associations among works, along with a user interface that allows rapid navigation across the links that distinguishes hypertext from other research that represents relationships within the computer.

Since we have defined hypertext in terms of intellectual works and links, these terms need definition too, provided in the sections below.

2.2.1 Work

Work. An artifact intended to create a communicative experience, such as a document, image, or song.

The notion of a work encompasses the range of things that can be linked together within a hypertext system. Since hypertext systems regularly link together various types of documents, such as texts, spreadsheets, CAD drawings, along with a variety of images, songs, etc., there is a need for a suitably broad term to encompass this diversity. Calling them all documents is unsatisfactory, since a song is clearly not a document. Describing them as objects is certainly broad enough, but is perhaps too broad, since such objects as rocks and plants have yet to be incorporated into any known hypertext system. The term node might appropriately be used in a hypertext context, tapping into the intuitive view of a hypertext as a mathematical graph. However, node seems unsatisfactory when applied to configuration management or document management systems, since it carries with it hypertext connotations that no longer apply.

When defining the notions of revision and variant, it will be useful to distinguish between the mental ideas and concepts concerning a work, and the fixation of those ideas as symbols in some medium. Consider an introductory text on set theory. The author of such a text has, in his head, a concept of what is appropriate material for the book. Topics such as definitions of a set, mappings, functions, relationships, etc. are all clearly in scope, and similarly, long excerpts from Shakespeare are just as clearly out of scope for such a volume. The inclusion of some advanced set theoretical concepts will depend on the individual author, and their goals and intent for the work. As the author works on the book, he will think about the contents (all within his understanding of the scope of the book), forming the ideas in his head that he wants to communicate. These ideas are internally translated into symbols such as words or set notation, and then fixed as symbols in some medium, such as a word processor file, or pen marks on a sheet of paper. This example leads to the following definitions:

Abstract work concept. The guiding idea for an intellectual work, the distinguishing essence that creates the borders of the internal idea space of the work.

Abstract work instance. A set of ideas and organizational frameworks, falling within the defining essence of a work concept, that together comprise the abstract content of an instance of the work concept. This instance is distinct from any particular fixation of the ideas into a set of symbols.

Symbolic work instance. An arrangement of symbols created through a mental process of fixing into a symbolic notation the set of ideas that compose an abstract work instance.

The goal of the "work concept" term is to act as a signifier for the essence of an intellectual work, the ideas that give it distinct identity across all revisions and variants.

The borders of a work concept are defined by concepts and test cases that guide the evaluation of work instances. A specific work instance is a revision or variant of a particular work concept if the ideas fixed within the work instance fall within the work concept's topology of ideas. A work concept is entirely abstract, is subjective, and is rarely communicated, typically inferred by analysis of concrete instances of the work. Observers frequently have differing notions of the work concept.

Consider the national anthem for the United States, the Star Spangled Banner. This has been sung at uncountable events, with subtle, and not-so-subtle variations in each performance. Yet, despite these differences, there remains a core nugget of Star-Spangled-Banner-ness, an archetype of note relationships and lyrics that leads us to recognize each performance as an instance of the same song. This essence is hard to define. Any effort to define the essence of the Star Spangled Banner prior to Woodstock would likely have excluded Jimi Hendrix's electric guitar version, even though it was clearly recognizable to the gathered audience. Trickier yet, would an audience have recognized the same performance if it had somehow been conveyed 50 years in the past, before the advent of the electric guitar? Furthermore, it's not obvious that an audience comprised of a general cross-section of the American population would recognize Hendrix's song as the Star Spangled Banner even today. Though the question is moot, it seems plausible that the Woodstock audience's understanding of the Jimi Hendrix's performance as a version of the Star Spangled Banner was predicated on their previous experience with electric guitar music. This prior experience opened mental pathways, created new understandings, and allowed them to recognize a version of the song played on an instrument unknown to Francis Scott Key, the original composer. Clearly the distinguishing essence of a work is subjective and malleable, capable of appropriation and reinterpretation by people other than the original author.

A work *concept* is distinct from the ideas that populate a work *instance*, and comprise its content. The work concept contains only those ideas and thoughts necessary to define the extent of the work. Content ideas fill in the conceptual space carved out by the work concept's guiding idea. The content ideas, like the work concept, are in the realm of the mental, and are distinct from any specific translation of the ideas into a set of symbols, such as a series of words, musical notation, or other graphical notation.

The author(s) of a particular work have some notion of abstract work concept for the work, and they use this to guide their development of the work. Other people who observe the work may, if they consider the issue, also develop their own understanding of the work concept. Even if they haven't considered it, they still have an implicit understanding of the work concept, based on their observation of the work. A reader of a textbook on Software Engineering would immediately view the addition of a fictional short story as being inappropriate to the scope of the textbook, even though he had never engaged the issue of the textbook's intellectual boundaries. Observers may agree with the understanding the authors have, but likely their understanding will diverge in some details. Unless the observer has put unusual energy into their consideration, the authors will have a more detailed model of the conceptual boundaries than any observer. One exception is creators of derivative works, who will likely have a sharp understanding of the work concept of the original, as well as of their additional modifications and contributions.

Even for the authors, understanding of the work concept will change over time, as the work is developed and refined. Reflection on the work will lead to new ideas being introduced and existing ideas being modified, and at times this process of refinement may stretch the boundaries of the work concept.

Introducing a mathematic notation for these concepts, we will use C_w to represent an abstract work concept, and I_w to represent an abstract work instance. Since these vary over time, *t*, and are relative to a particular person, *p*, they are described as:

$C_w(p, t)$ – Person *p*'s perception of the abstract work concept at time *t*.

$I_{w,n}(C_w(p, t), p, t)$ – One of n abstract work instances person *p* associates with $C_w(p, t)$ at time *t*. Each instance at a given time represents a variation in the ideas that comprise the content of the work.

There are many symbolic notations, such as words and numbers, musical notations, engineering drawings, geographic maps, entity-relationship diagrams, control flow diagrams, programming language statements, etc. In all of these notations, the symbols *represent* specific ideas, abstractions, or concepts. That is, they are not the ideas themselves, but symbolic stand-ins for the ideas. Using the terminology of semiotics, they are signifiers, as distinct from the signified. Note that the symbolic work instance is media-independent. Symbols in the symbolic work instance are abstract, and have not yet been realized in a specific persistent storage medium, such as words on paper or in a word processing file, or, notes on magnetic media or in musical notation.

A particular symbolic work instance is denoted:

$S_{w,m}(I_{w,n}(C_w(p, t),p,t),p,t)$ – One of *m* symbolic work instances into which person *p* has fixed the ideas from $I_{w,n}(C_w(p, t), p, t)$ using symbolic notation, at time *t*.

2.2.1.1 Computer Representation

Unlike the abstract definition of a work in the previous section, this section describes the process of how the internal (within a person's mind) symbolic representation of a work is translated into a specific computer representation, and vice-versa, how the computer manipulates information so a person can construct an internal set of symbols and ideas from it.

> *Symbolic rendition.* A computer-mediated representation of a data item, or representation of the output of a computational process, as symbols.

As a computer-generated symbolic representation, the symbolic rendition is an important bridge between an individual's symbolic work instances in the conceptual layer, and their concrete representation within a computer. The act of authoring involves the representation of the symbolic work instance as a symbolic rendition. The act of reading involves the creation, or refreshing of a symbolic work instance, and the subsequent creation or refreshing of ideas in the abstract work instance.

A particular symbolic work instance, a collection of symbols in some person's mind, can be represented by many possible renderings. For example, a simple boxes and arrows diagram could be represented as a graphical display by a dedicated drawing program such as Visio, which would be slightly different from a representation of the same diagram using the drawing features integrated within a word processor, which would be different still from an ASCII art representation using simple characters in a text editor.

When the symbolic representation is a rendition on a computer screen, it is often termed a *view*. However, the notion of a symbolic rendition spans multiple input and output devices, and encompasses such symbolic representations as the playing of a music file, and output in Braille notation, neither of which seem well-described by the term "view".

A symbolic rendition of a work is denoted as R_w. The operation of translating the symbols in a symbolic instance into a symbolic rendition is denoted by the function T_s, and hence:

$R_w(t) = T_s(S_{w,m}(I_{w,n}(p,t),p,t))$

The act of translating the symbolic work instance renders it into symbols that are capable of being interpreted by many people, and hence the symbolic rendering is not dependent on a particular person. However, the symbolic rendering does vary over time, as would be expected due to the varying of the symbolic work instance over time. This represents the modification of a work over time, as it is authored or maintained.

> *Renderer:* a computational process that creates and mediates interactions with a symbolic rendering.

For computer media, a computational process, called the renderer, is responsible for creating and maintaining a symbolic rendering. In the case where the rendering is to a visual display, the renderer is also known as a *viewer*. The renderer handles input from the user concerning navigation within, or modifications to the rendition.

When it creates a rendering, the renderer acts upon either persistently stored data, a representation of the persistently stored data, or the output of a computational process. Most systems use the persistently stored data without modification. However, systems like the Web that place a client-server split between the renderer (Web browser) and the server, it is possible that the server may perform some processing on the data item, or output of a computational process, before it is transmitted to the renderer, a process known as representational state transfer [69].

Renditions may be read-only, or modifiable. If they can be changed, modifications to the rendition result in modifications to the data item, or the (output of the) computational process.

The process of creating a symbolic rendition is denoted as T_r. An object is denoted as $D(t)$, and a computational process is $P_c(t,i)$, for some set of input i, at time t. The raw output of a computational process is denoted $D_c = P_c(t,i)$. The operation of creating a representation of the data for representational state transfer is W_t. Thus, the data acted upon by the renderer is either $D(t)$, D_c, or $W_t(D(t))$, $W_t(D_c)$. Regardless of source, the data acted upon by the renderer will be denoted as D_{render}, and hence:

$D_{render}(t) = D(t) \mid D_c \mid W_t(D(t)) \mid W_t(D_c)$ with “|” standing for “or”

The symbolic rendition can now be described as:

$R_w(t) = T_r(D_{render}(t))$

Combining with the equation for creating a symbolic rendition from a symbolic work instance yields:

$T_s(S_{w,m}(I_{w,n}(C_w(p,t),p,t),p,t)) = R_w(t) = T_r(D_{render}(t))$

This highlights the role of the symbolic rendition as a bridge between the personal, mental mapping of ideas into symbols, and the mechanical mapping of data into symbols as performed by a computer.

Figure 1 below shows the model of a work. The top of the figure, the conceptual level, shows those aspects of a work that are within the mind, as abstractions, ideas, and symbols. The bottom half of the figure, the computer representation level, shows the representation of the work within the computer.

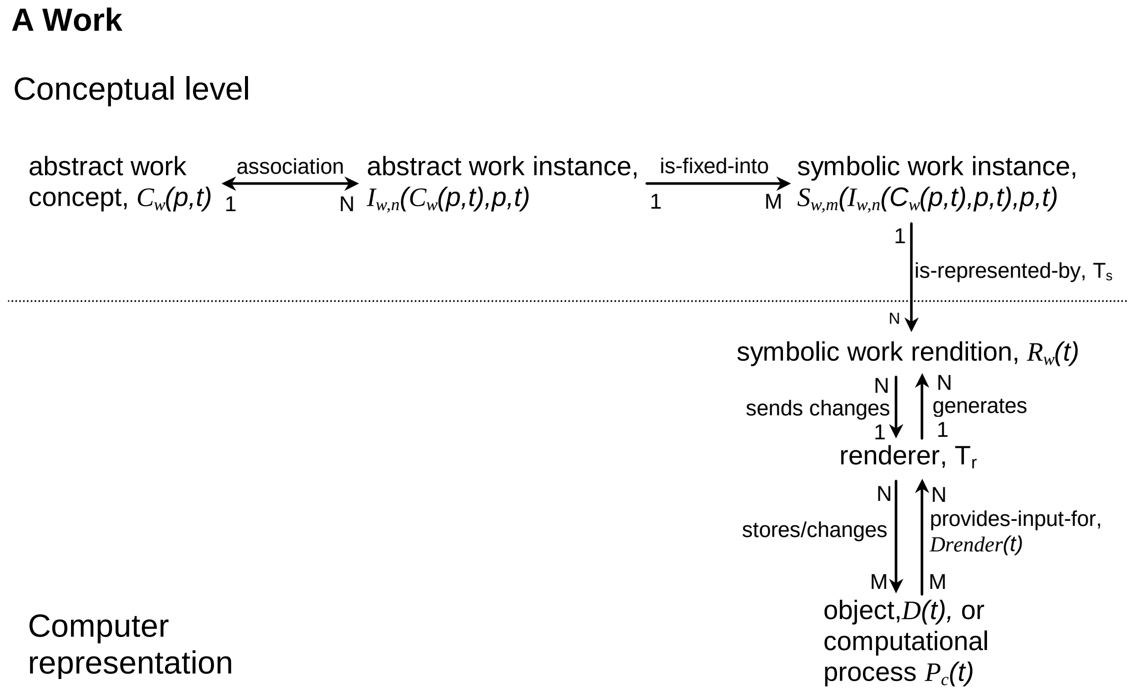

A Work
Conceptual level
abstract work concept, $C_w(p,t)$
association
1
N
abstract work instance, $I_{w,n}(C_w(p,t),p,t)$
is-fixed-into
1
M
symbolic work instance, $S_{w,m}(I_{w,n}(C_w(p,t),p,t),p,t)$
1
is-represented-by, T_s
N
symbolic work rendition, $R_w(t)$
N
sends changes
1
N
generates
1
renderer, T_r
N
stores/changes
M
N
provides-input-for, $Drender(t)$
M
object, $D(t)$, or computational process $P_c(t)$
Computer representation

Figure 1 – An intellectual work, at the conceptual level, within the mind, and the computer representation level.

2.2.1.2 Alternate Terms

There is a wide diversity of terms used to describe intellectual works within hypertext systems, as highlighted in Table 1. In this table, when there were multiple references for a system, or the system underwent several revisions during its development, it uses the earliest reference. Systems shown in the table are representative, and do not constitute an exhaustive list.

These terms can be divided into several categories, as described below.

Some systems simply use the term *document* for linkable information, with Xanadu [137], Intermedia [205], and Microcosm [72] being examples. In the case of Intermedia, the use of document encompasses text, two and three-dimensional graphics displays, and timelines, and in Microcosm, it has a similarly broad range, spanning word processing documents, spreadsheets, CAD drawings, images, and many other content types. However, interactive graphical displays do not typically come to mind for the term document, which carries with it static connotations. This leads to a search for alternate terms that are more appropriate and encompassing.

Other systems name the linkable information after their user interface metaphor. Both HyperCard [8] and NoteCards [189] use the term *card*, or *notecard*, consistent with their card-based user interfaces. KMS [3] uses the term *frame* to represent the information that fits onto a single screen, with the metaphor being that KMS acts as a “frame” around that information. The Virtual Notebook System (VNS), uses a notebook metaphor, with information subdivided into *pages*, and the Hyperties system uses an encyclopedia metaphor, organizing information into *articles*.

Early on, hypertext researchers recognized that mathematical networks could model a set of linked documents [60], and hence hypertext systems such as Neptune [45], Sun’s Link Service [147], HyperProp/NCM [179], and CoVer [87] (to name just a few) employ the terminology of networks by calling their information items *nodes*. This is a useful abstraction for hypermedia systems, since the term node carries neutral connotations about the kind of information within the node, and hence it can equally model documents, images, movies, CAD drawings, or any other kind of content. Another term that has similarly broad connotations is *object*, employed, for example, by Aquanet [126], PROXHY [106], HyperForm [202], and Chimera [7]. For Aquanet, PROXHY, and HyperForm, the use of object accompanies object inheritance and/or message passing to objects within the system, while Chimera just uses the term abstractly to refer to any data item, without object-oriented programming semantics. Avoiding node and object, the Dexter model [94], and hence the DHM system [85], uses the term *component*.

System	Date	Term for intellectual work	Term for link endpoint
NLS [60]	1968	file, statement (a file is a collection of statements)	location or name of a statement
Xanadu [137]	1981	document, span (a document is a collection of spans)	span
TEXTNET [191]	1986 (1983)	chunk (also toc, for table of contents node). Also uses node.	none (link is to an entire chunk)
Neptune [45]	1986	node (document is a set of nodes)	character position, or span
NoteCards [189]	1986	node, notecard	link icon
Hyperties [172]	1987 (1983)	article (uses encyclopedia metaphor)	embedded menu
KMS [3]	1987	frame (screen-sized)	linked item, link source
WE [177]	1987	(hypertext) document, node (a document is a collection of nodes)	N/A (links go to/from entire nodes)
HyperCard [8]	1988	card (each card is associated with a background)	button
Intermedia [205]	1988	document	block (but also discusses anchoring)
Sun's Link Service [147]	1989	node, linkable object	link indicator (an icon, or glyph)
Virtual Notebook System (VNS) [170]	1989	page	link (the term link is used for both the link, and its endpoint, depicted by a small icon)
HOT/eggs [157]	1990	node	link endpoint, hotspot
Microcosm [72]	1990	document	selection
Dexter model [94]	1990	component, atomic component	anchor
World Wide Web [22]	1990/2	resource	anchor
HyperProp/NCM [27]	1991	node	anchor
ABC/Artifact-Based Collaboration [176]	1991	node, data object, artifact	anchor, position in data
Aquanet [126]	1991	basic object	graphic element (describes layout of a relation, not linking within text)
PROXHY [106]	1991	object, node (nodes contain objects)	anchor
HyperPro [141]	1992	node	anchor is used in the paper, but HyperPro has no anchor support.
CoVer [87]	1992	node	link anchor
Multicard [162]	1992	node	anchor
DHM [85]	1992	component	anchor
Hyperform [202]	1992	object	N/A (framework could be used to create an anchor object)
Chimera [7]	1994	object	anchor
Dolphin [92]	1994	node (contents are links, scribbles, text, images, and other nodes)	none ("Links present themselves as arrows with a handle." p. 7)
HyperDisco [204]	1996	node (composites can also be linked)	anchor
HyperStorM [15]	1996	node (AtomicNode, Composite-Node, VirtualCompositeNode)	none (basic link types are only node to node)

Table 1 – Hypertext systems, and their term for linkable information, and link endpoint.

Outside the realm of hypertext systems, a variety of terms are used to describe the representation of intellectual works within the system. The Document Management Alliance 1.0 [50] specification uses the term *document version object*, which contains a *rendition object*, which in turn contains a *content element object*. The Portable Common Tool Environment (PCTE) [195] uses the term *object*. Within versioning and configuration management systems, SCCS [163] uses the term *module*, containing multiple *revisions*, while CVS uses the term *file* [20], Adele [61] uses *object*, and NUCM [192] uses *artifact*. In their configuration management systems survey, Conradi and Westfechtel use the term *software object* [35].

The term "work" as used herein has been borrowed from copyright law [182]. Copyright law and hypertext research have both struggled with the same problem, that of finding a suitably broad term that encompasses the wide variety of artifacts that, when viewed, read, heard, or watched, communicate ideas. While hypertext researchers have struggled to abstract away from their early focus on text and documents, copyright law has long used the term "work" to signify a broad range of intellectual communicative artifacts, irrespective of medium.

2.2.2 Anchor

Anchor. A handle for a specific set of symbols within a work instance.

While systems such as TEXTNET [191], WE [177] and HyperStorM [15] only support links that connect an entire object to another whole object, most provide for attaching the link to a specific endpoint within the object contents. Anchors typically exist in relationship to a work. That is, an anchor acts as a handle for a specific subset of a work's symbols, and is depicted as part of the work's symbolic rendition. For example, in a text, anchors are rendered as either underlined or highlighted words, or as an icon embedded in the text. This pictorially represents that the anchor is acting as a proxy, or handle for the underlined or neighboring words (symbols). Defining an anchor as a handle for specific symbols of a work allows the anchor to exist independent of any particular symbolic rendition of the work, and hence the same anchor can exist across multiple renditions, and can be depicted in multiple ways within a given rendition.

If an anchor's representation within a computer no longer causes the correct words or objects to be highlighted, a situation that can occur when anchors are not stored with a work, the definition of the anchor has not changed. Instead the computer representation of the anchor is inconsistent with its conceptual definition. If the anchors are defined as a region, they have only an implicit association with the symbols represented in that region, since the region explicitly identifies a space, not a set of symbols. The anchor as region definition does not explicitly imply that a change in the symbolic representation of a work must result in a change to the anchor's region. This connection is implicit, hidden in the assumption that the region encloses meaningful symbols.

The same abstractions used to describe a work at both the conceptual and the concrete level, apply as well to anchors. Though conceptually not nearly as large as an intellectual work, anchors do have an associated abstract concept that provides a defining abstraction giving the abstract boundaries of the anchor. Similar to the term "abstract work concept", the defining abstraction for an anchor is known as an "abstract anchor concept". The author of an anchor associates a set of ideas with them, thereby providing the abstract intellectual content of the anchor, known as the "abstract anchor instance." The ideas in the abstract anchor instance are fixed into symbols in the "symbolic anchor instance." The computer then represents the conceptual anchor as a symbolic annotation to a work rendition.

Expressing an anchor using mathematic notation, an anchor's abstract work concept is denoted $C_a(p,t)$, and abstract anchor instances are $I_{a,n}(C_a(p,t),p,t)$. Since the anchor is associated with the symbols of a work, the symbolic anchor instance is denoted as:

$S_{a,m}(I_{a,n}(C_a(p,t),p,t), S_{w,m}(I_{w,n}(p,t),p,t)))$

That is, the symbolic anchor representation depends on the ideas defining the anchor, $I_{a,n}$, and a specific symbolic work instance $S_{w,m}$, for a given person, p, at time t. Anchors are almost always rendered along with the work, and so a work rendition in the presence of anchors is:

$R_w(t) = T_s(S_{w,m}(I_{w,n}(p,t),p,t)) + T_s(S_{a,m}(I_{a,n}(C_a(p,t),p,t), S_{w,m}(I_{w,n}(p,t),p,t))))$

Or, more simply, the symbolic rendition of the work is the symbolic union of the transformation of the symbolic work instance, $T_s(S_{w,m})$, and the symbolic anchor instance, $T_s(S_{a,m})$. This is a different, but equivalent notion to the Chimera concept of an anchor being defined on a view [7].

A first class anchor within a hypertext system is represented as an object, $D_a(t)$, or a computational process, $P_{c,a}(t)$. After potentially experiencing representational state transfer, the anchor information arrives at the renderer as $D_{render,a}(t)$. The rendering of the work representation is denoted:

$R_w(t) = T_r(D_{render}(t)) + T_r(D_{render,a}(t))$

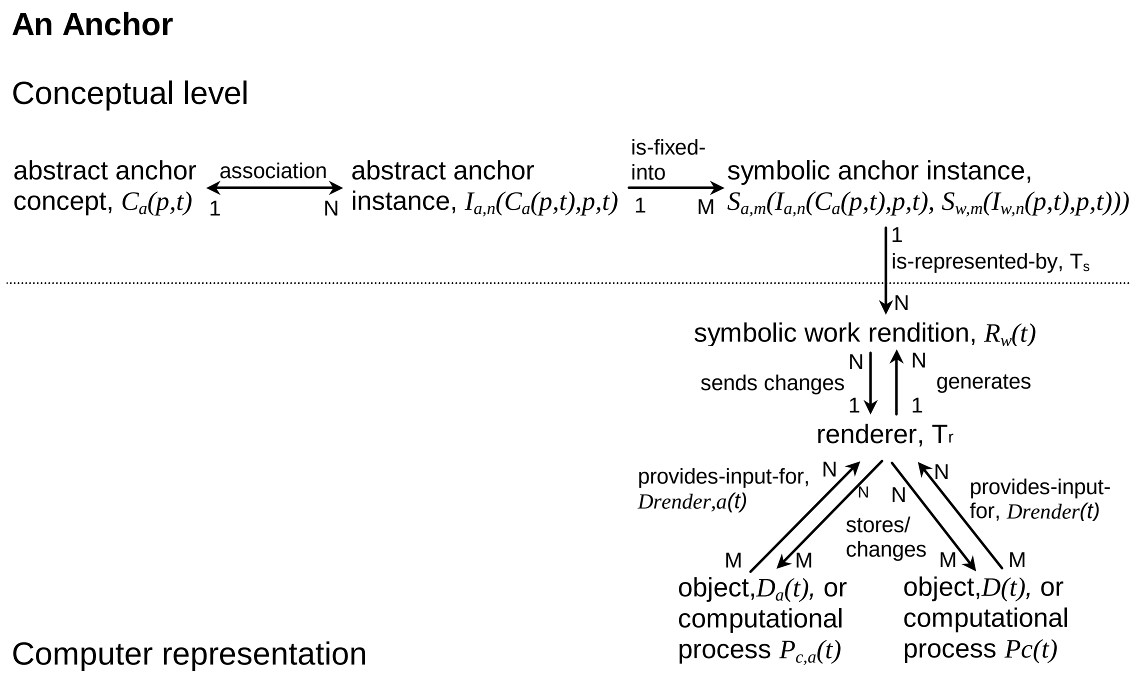

An Anchor
Conceptual level
abstract anchor concept, $C_a(p,t)$
association
1
N
abstract anchor instance, $I_{a,n}(C_a(p,t),p,t)$
is-fixed-into
1
M
symbolic anchor instance, $S_{a,m}(I_{a,n}(C_a(p,t),p,t), S_{w,m}(I_{w,n}(p,t),p,t)))$
1
is-represented-by, T_s
N
symbolic work rendition, $R_w(t)$
N
N
sends changes
generates
1
1
renderer, T_r
provides-input-for, $Drender,a(t)$
N
N
N
stores/ changes
N
provides-input-for, $Drender(t)$
M
M
M
M
object,$D_a(t)$, or computational process $P_{c,a}(t)$
object,$D(t)$, or computational process $Pc(t)$
Computer representation

Figure 2 – The conceptual model of an anchor, and its computer representation.

If the anchor information is imbedded within $D_{render}(t)$, then the rendering of the work also renders the anchors:

$$R_w(t) = T_r(D_{render}(t))$$

2.2.2.1 History of the Anchor

Early hypertext systems used a variety of terms to describe a link endpoint. Some systems use the term for the address or location of the endpoint, such as *location* or *statement name* in NLS [60], and *character position* in Neptune [45]. Others emphasize the user interface element that was used to activate a link traversal, as with *link icon* in NoteCards [189], *embedded menu* in Hyperties [172], *button* in HyperCard [8], *link indicator* in Sun's Link Service [147], and *selection* in Microcosm [72].

Coined by Norm Meyrowitz [130] in the mid to late 1980's, and initially used by the Intermedia group, the term *anchor* came to represent both the endpoint reference as well as the user interface representation of a link endpoint. The motivation for developing the term anchor was similar to that for node and object: generality of the abstract concept of a link endpoint across multiple content types. In the case of Intermedia, the term *block* was initially used, as in a block of text. However, once images were introduced, it was reasonable to consider non-square, and hence non-block-like link endpoints, and thus this led to a desire for a more abstract term. One coined, the term anchor spread first to other researchers in contact with the Intermedia group. At the initial Dexter workshop, the need for anchors was strongly debated, with the Dexter group in the end agreeing for its need and adopting the term [114]. From the Dexter participants, the term spread to the wider hypertext community at the Hypertext'89 conference, with Norm Meyrowitz using the term in his keynote address [129], and with Frank Halasz using the term in two separate sessions on NoteCards, and the Dexter model [140]. As is visible in Table 1, by 1991 the term anchor was almost universally adopted by the hypertext community.

2.2.3 Link

Link. An association among a set of work instances, a set of anchors, or their combination.

In set theory, a binary relation is a statement $R(x, y)$ that is either true, or not true for each ordered pair of elements in a given set A. A *representing graph*, of a relation in A is a graph, G, consisting of all ordered pairs (x, y) from A for which $R(x, y)$ is true [151]. If the set A is taken to consist of all of the linkable work instances within a hypertext system, then an individual, static, hypertext link can be viewed as a member of the representing graph for the link's relation. Since each member of G is an ordered pair, that is, two members from A, it is possible to view each single ordered pair as an ordered *set*. For systems that allow links to be bi-directionally navigated, this set does not need to be ordered.

But, set of what? It would be easy to say that hypertext links are a set of work instances, but this would imply that links could not directly connect to an exact set of symbols within a work, such as a specific word or part of an image. However, many hypertext systems do provide links to within-work destinations. If x is an object, and p is a set of symbols within a work instance, the anchor provides an (x, p) pair, thus allowing a subset of a work instance's symbols to be specified. The anchor allows a *link* to be defined as a possibly ordered set of anchors. The link is unordered when it is bi-directional, or contains just a single anchor.

However, links were initially taken to be members of the representation graph, G, for relation, $R(x, y)$, among work instances, not anchors. The switch to using anchors requires a similar switch to define the relation, $R_a(a_1, a_2)$, as being over a set of anchors, A_a. Each link is an ordered pair that is a member of the representing graph for the anchor relation.

For systems, like Chimera, that support n-ary links, these definitions can be extended to cover non-binary links. An n-ary link is still a possibly ordered set of anchors, but now the set is not limited to just two members. A link of n anchors is a point that satisfies a relation of n elements.

The significance of defining a link as a set is that it *allows static links to be represented within a computer using a container*. In particular, any of the container implementations given in Section 4.2 could be used to represent a link. It also simplifies analysis of hypertext systems considerably. Whereas links and containers are usually considered separately, now there are just containers, which can model composite objects, collections of objects, and links. Any versioning mechanism that applies to containers equally applies to composites, collections, and links. Furthermore, it elevates containment relationships between system entities to a primary concern within hypertext systems, since it is these relationships that dominate their data models.

The notation developed for describing a work also extends to a description of links:

- The abstract link concept, $C_l\,(p,t)$, depends on the person perceiving the abstraction, and the time this perception was held. It provides a defining abstraction giving the abstract boundaries of the link, typically the link's association, and rules for evaluating whether work instances or anchors have that association.
- The abstract link instances, $I_{l,n}(C_l\,(p,t),p,t)$, depend on the abstract link concept, as well as a person and time. For a link, it includes ideas concerning the work instances and anchors that have been found to have a particular association, along with the ideas that fill in the content of the link, such as annotation links.
- The symbolic link instance, $S_{l,m}(I_{l,n}(C_l\,(p,t),p,t),p,t)$, depends on a specific abstract link instance, for a specific person and time. It is the fixation of the link's content as symbols.

A symbolic link rendition, $R_l(t)$, is a symbolic representation focusing on just representing the links. This is the link, or network view of hypertexts available in many systems. Links are also represented as part of a work's symbolic rendition, $R_w(t)$, such as by an icon the represents link between work instances. If the link is between anchors, then typically it is the anchors, and not the link that is represented within the work's symbolic rendition. A transformation function, T_s, transforms the symbolic link instance at the conceptual level (within the mind) into a rendition in the concrete level (within the computer). Thus it is possible to write:

$$R_l(t) = S_{l,m}(I_{l,n}(C_l(p,t),p,t),p,t)$$

and

$$R_w(t) = T_s(S_{w,m}(I_{w,n}(C_w(p,t),p,t),p,t)) + T_s(S_{l,\mathrm{m}}(I_{l,\mathrm{n}}(C_l(p,t),p,t),p,t))$$

where the "+" operator indicates a symbolic combination, or union, of the symbolic renditions of the work and the link.

A *static link* is one where the work instances or anchors are explicitly known and represented within the computer. When a link is a first class abstraction in a hypertext system, the persistent representation of the information used to create the link rendition is an object, $D_l(t)$. A *dynamic link* is one where only the relation is known, but the explicit work instances or anchors are known only by performing a computational process, $P_{c,l}(t)$. A dynamic link is modeled as a computational process that can produce, for given a_1, the set of all ordered pairs (a_1, a_2), denoted $D_{c,l}$, that satisfy the link's relation $R_a(a_1, a_2)$. Repeating this process for all a can generate the entire representing graph for the relation.

A computational process called a renderer, $\mathrm{T_r}$ creates and mediates interactions with link and work renditions. If the link information is being retrieved over a network connection, it can possibly be modified before transmission, and hence this operation of creating a representation of the object for representational state transfer is denoted W_t. Thus, the data acted upon by the renderer when creating a link rendition is either $D_l(t)$, $D_{c,l}$, or $W_t(D_l(t))$, $W_t(D_{c,l})$. Regardless of source, the link data acted upon by the renderer will be denoted as $D_{render,l}$, and hence:

$D_{render,l}(t) = D_l\,(t) \;|\; D_{c,l} \mid W_t(D_l(t)) \;|\; W_t(D_{c,l})$ with "|" standing for "or"

A link rendition can now be described as:

$$R_l\,(t) = T_{r,l}(D_{render,l}(t))$$

A rendition that symbolically represents both link and work information can be described as:

$$R_w(t) = T_r(D_{render}(t)) + T_r(D_{render,l}(t))$$

That is, it is the combined rendition of the work data representation and the link data representation.

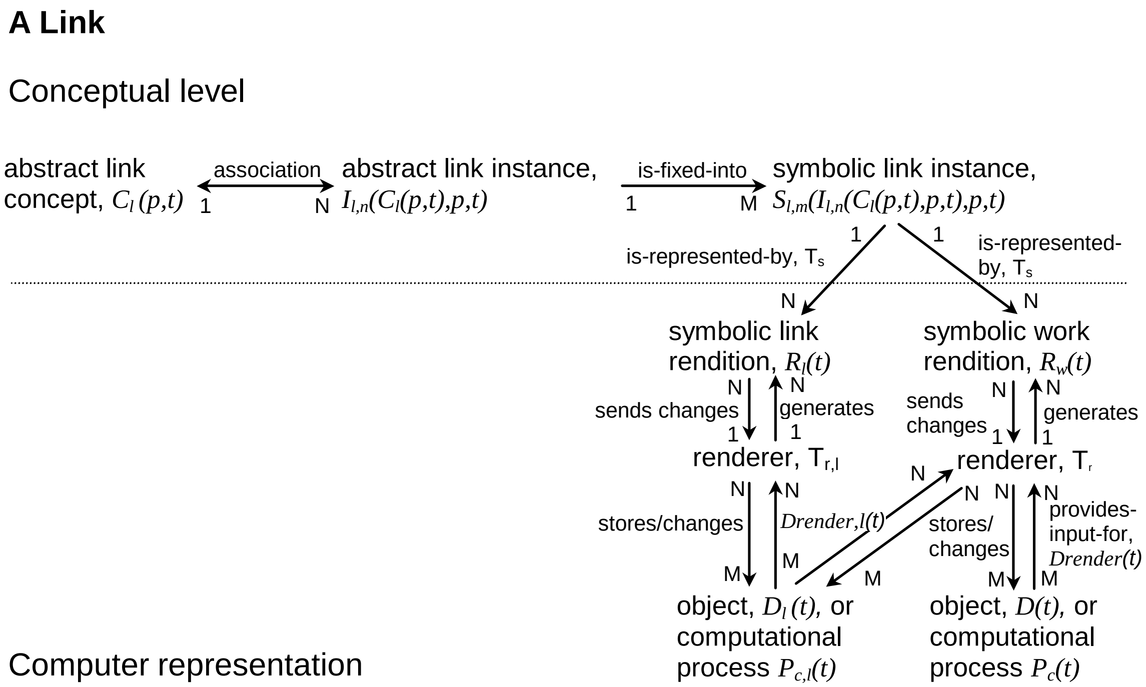

A Link
Conceptual level
abstract link concept, $C_l(p,t)$
association
1
N
abstract link instance, $I_{l,n}(C_l(p,t),p,t)$
is-fixed-into
1
M
symbolic link instance, $S_{l,m}(I_{l,n}(C_l(p,t),p,t),p,t)$
1
1
is-represented-by, T_s
is-represented-by, T_s
N
N
symbolic link rendition, $R_l(t)$
symbolic work rendition, $R_w(t)$
sends changes
generates
sends changes
generates
N
N
1
1
renderer, $T_{r,l}$
renderer, T_r
stores/changes
$D_{render,l}(t)$
stores/ changes
provides-input-for, $D_{render}(t)$
M
M
object, $D_l(t)$, or computational process $P_{c,l}(t)$
object, $D(t)$, or computational process $P_c(t)$
Computer representation

Figure 3 – The conceptual model of a link, and its representation within a computer. The link can either be rendered into a link only rendition, $R_l(t)$, or it can be integrated into the rendition of the work, $R_w(t)$.

2.2.3.1 History of the Link

In his 1945 essay, "As We May Think," Vannevar Bush described the Memex, a machine that would allow its user to capture arbitrary *associations* between documents stored within the system, though in all examples used in the paper, associated documents always share a related subject [25]. Bush used several wordings for these associations, describing them as "*tying* two items together," "two items to be *joined*," and "*binding* items together into a new book." Joining items together forms a "trail," or "associative trail," and items can be "*linked* into the main trail," and thus a trail can be viewed as a collection of associations (emphasis added, all quotes from p. 107-108 of [25]). In Bush's article, the items being associated are not in question: they are all documents. However, the terminology of association was clearly in flux, as any of the terms association, tie, joint, binding, or link could equally be used, though only association was ever used in its noun form, all others being used as verbs.

The early pioneers of hypertext were all directly influenced by Bush's vision, but reinterpreted it to employ digital computers instead of microfilm to represent the documents and their associations. Indeed, Engelbart directly acknowledges Bush in a 1962 letter to him seeking permission to quote from "As We May Think" for an early report for the NLS project [59], and a 1972 paper by Nelson titled, "As We Will

Think," is a detailed retrospective on Memex, as interpreted by the Xanadu system [138]. Early on, there was agreement that the computer representation of an association is called a *link*, with NLS using the term in 1968 [60], and Xanadu in 1972 (and probably earlier) [138], and from these beginnings there has been broad acceptance and use of the term link. The alternative terms tie, joint, and binding were never adopted, though the term "association" is still used today. The notion of a guided trail has also survived to the present, and continues to be an active area of research (e.g., the work on Walden's Paths in [171]).

Grønbæk and Trigg describe four classes of link styles in hypertext systems [86], p. 70-73:

- **Links as addresses:** The address of the link destination is embedded within the work. Examples include NLS, HyperCard [8], and the World Wide Web.
- **Links as associations:** Links are first-class objects that express an association between works. Link traversal is two-way, and can be initiated from either endpoint. Examples include Intermedia [205], Chimera [7], SEPIA [181], and many others.
- **Links as structural elements:** Links are used to represent hierarchical, or other organization of materials. When used to represent hierarchical containment structure, this use of links is one of several possible representations of containment relationships. The fileboxes in NoteCards [189] are an example of this use of links.
- **Links for rhetorical representation:** Links represent the structure of an argument. gIBIS [34], Aquanet [126], and the Author's Argumentation Assistant [169] all use links to represent argumentation structure.

To these link styles can be added:

- **Links as semantic network:** Link types are used to represent semantic relationships between works, and may not be intended for link navigation. MacWeb [135] is one system that exemplifies this style.

Across all these styles of link types, the link expresses the existence of a relationship between the linked works. Even when the link is simply an address, the link was created on purpose, to express that the works are connected in some way. Papers have described possible kinds of link relationships [191], and taxonomies of link relationships [48].

2.3 Version

Unversioned object. An object that has only one state, the current state, and modifications overwrite it.

Revision. A snapshot of an instant in the evolution of a work or entity.

Versioning. The act of recording the evolution of a work or entity, using revisions to represent explicit states in the evolution.

The typical user of today's computers works with files that are not under revision control. For example, unless some additional version control facility is employed, all files in the Unix, Windows, and MacOS operating systems are unversioned. More precisely, an *unversioned object* is an object that has only one state, the current state, and modifications overwrite it.

People use "version" to mean many things. A version can represent a modification to the content of an item, as in "the version from last Tuesday," and these modifications can include those made by the original author(s), or even by different authorities. For example, [122] mentions that, "a hypertext edition of *The Waste Land*, for instance, would enable comparison of T. S. Eliot's original draft, Pound's corrected copy, and the final published version" (p. 303). Version is also used to represent a mechanical change to an item without changing its meaning, as in the "PDF version of the document." In the context of source code development, version can also mean a change made to accommodate a different platform, operating environment, or feature, as in "the MacOS version" or the "debug version" of a source code file. Version can also mean a natural language translation, as in "the German version of the document." Finally, there is the notion of a version that captures intellectual precursors, even though the derived work is different enough to have a separate identity and history, as in the Jimi Hendrix version of the Star Spangled Banner, or the King James Version of the Bible.

Providing computer support for these different uses of the term version implies different features. For example, recording different versions of a data item involves different operations from mechanically deriving a different version of an existing item. In order to separate these different features, the configuration management literature has developed the distinct terms *revision* and *variant* for these two senses of "version" [35].

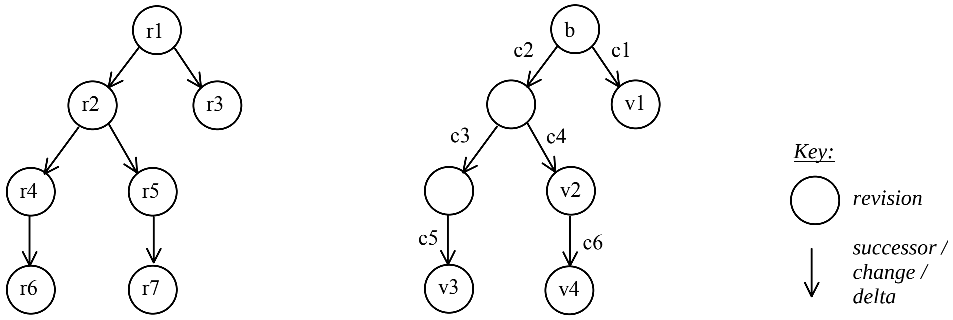
While conceptually a revision is a snapshot in the evolution of a work or entity, Figure 1 shows that an instance of a work at a given time, *t*, has at least one symbolic work rendition, $R_w(t)$, created by rendering the data stream $D_{render}(t)$, which is one of $D(t)$, D_c, $W_t(D(t))$, or $W_t(D_c)$. Thus, a revision persistently stores either the object $D(t)$ or the process description, $P_c(t)$, that creates D_c, since these are the base items which are used to create the symbolic work rendition that is being captured as a revision. If a revision control system is not in use, revisions can be maintained by storing each snapshot of $D(t)$ or $P_c(t)$ in a separate unversioned object. This is a common practice for people collaborating on documents via email, where successive revisions of the document are stored in separate unversioned files. Even though these documents are not managed by a revision control system, the document's authors still view them as distinct, separate revisions. When a computer takes over the management of revisions, it provides advantages such as automatically recording the predecessors and successors of each revision, providing identifiers (e.g., "1.1", "Beta1") for each revision, and enforcing policies such as prohibiting the modification of older revisions.

Revision histories are represented using a graph structure, where the nodes of the graph are revisions, and the arcs are predecessor and successor relationships. In the simplest case, known as a *linear version history*, the graph is a straight line, where no revision has more than one predecessor. When a revision may have more than one successor, but only one predecessor, then revisions form a *tree version history*. This is used when different branches of the tree represent variants. If it is possible to merge branches together, thereby allowing revisions to have more than a single predecessor, then the revisions form a *directed acyclic graph (DAG)*. This occurs when branches are used to represent different developments by collaborators working simultaneously (in parallel), and the contributions of each collaborator are merged together.

2.3.1 State-based and Change-based Versioning

The kinds of works and entities that are versioned in hypertext versioning systems include anchors, links, documents, composites, and contexts. The evolution of these items can be viewed in two ways, as either a set of instances, or as a set of changes between instances. These are known as *state-based* and *change-based* versioning respectively [35]. RCS [185] and SCCS [163] are classic examples of state-based systems, while PIE [80] and EPOS [119] exemplify the change-based orientation. Since most state-based systems employ concrete representations that persistently store the difference between successive states, called a *delta*, it is worth asking what is the fundamental difference between state-based and change-based systems. An essential difference is whether the states, or the changes are named, and hence first-class objects in the system. In state-based systems, the states are named with revision identifiers and the deltas are unnamed, while in change-based systems it is the changes that have identifiers, and intermediate states between revisions are anonymous. Figure 4 highlights the difference between state-oriented and change-oriented versioning. In Figure 4a, all of the states are explicitly named revisions, while the changes have no separate identity. In Figure 4b, all changes are explicitly named, and several unused intermediate states are anonymous. The first state of Figure 4b is the initial, or baseline state, to which all changes are applied. Revision v1 is constructed by applying change c1 to the baseline, and revision v3 is constructed by applying changes c2, c3, and c5 in order to the baseline. As Figure 4 highlights, the type of graph structure formed by the predecessor and successor relationships is orthogonal to whether the system is state-based or change-based.

As a further nuance for state-based versioning, the Palimpsest [56] and VTML [194] delta formats are designed to record the changes between revisions in a state-based system, yet maintain a fine-grain history of the changes between states, recording down to the byte level exactly who made each change. Durand describes the set of operations that record these modifications as being *change complete* [56]. This is in contrast to the more typical deltas employed by, for example [104,78,46], which do not record the exact set of steps performed to go from one revision to the next, using a set of operations that have the property of being *version complete* [56]. As an example of the difference between the two styles, when two revisions from parallel branches are merged using VTML, the system explicitly records, with the MERGE tag, which individual changes belong in the merged revision, capturing exactly the changes, and their cause – a merge operation. In contrast, traditional deltas only record insertions and deletions of whole lines, do not record semantic information, like merging, and make it difficult to reconstruct who made a particular change.



a) State-oriented versioned item *b) Change-oriented versioned item*

Figure 4 – A revision history shown with state and change orientation.

2.3.2 Variant

> *Variant.* A snapshot of an instant in the evolution of a work or entity, whose differences from other snapshots can be precisely specified, or parameterized, in a form other than a delta.
>
> *Rendition.* A mechanically derivable variant.
>
> *Alternate version.* A variant that is sufficiently different from other instances of a work or entity that causes it to have a new abstract work (or entity) concept, hence a change in identity.

A variant is a computer-maintained record of a work that, while falling within the definition of commonality of an abstract work concept, differs from other instances of the same work in well-understood ways. This is similar to Tichy's definition of variants as objects that are indistinguishable under a given abstraction [186].

There is an inherent tension in the notion of variance. An abstraction is bound together by crisply defined conceptual boundaries that capture the essence of *sameness* among instances of the abstraction. But, by its nature, a variant expresses *difference* among these similar instances. So long as the differences are small, and stay within the conceptual boundaries, variants are not troublesome. But, once the variances test the conceptual boundaries of the abstraction, it becomes harder to tell whether a specific instance is a variant of one abstraction, or another. As an appeal to intuition, ponder the kinds of chairs one encounters in the furniture sections of art museums, whose forms often test the boundaries of the essence of what is considered a chair. Are they variants upon a chair, or are they just abstract sculptures? The placement of these chairs in a museum begs the question.

Consider a module of computer source code. As this source code is written, the author may decide to save the current contents of the module at a particular point in time, thus creating a revision of the module. The author may also modify the module so it will use the capabilities of a different operating system. Since the module behaves the same, despite the change in operating system, the new module still meets the abstract criteria for what belongs inside the module, and hence it is a variant of the module. Since the change was made to accommodate a specific operating system, this information specifies how it differs from other revisions and variants.

This introduces the notion of parameterization of variants, and variation along a specific *axis of variability*. An axis of variability represents a set of related differences that differ by the value of a single parameter. As shown in this example, the operating system often forms an axis of variability, with specific operating systems forming points along this axis. When a variant can be mechanically derived, it is termed a *rendition*. For example, PDF and HTML variants can be mechanically generated from most word processing documents, and thus are considered renditions of the original document. In this case, they are variants along the document format axis of variability.

Interestingly, the definition of variant encompasses the notion of revision too, since, like a variant, each revision is also snapshot of an instance of a work or entity. Revisions can be viewed as variants that have two axes of variability, the revision's creation time, and, when the revision history includes branches, the revision's branch. In general, the interactions between revisions and variants of the same work can be quite complex, as demonstrated by the following quote from the Sixth Edition of Darwin's The Origin of Species [40]:

> As copies of the present work will be sent abroad, it may be of use if I specify the state of the foreign editions. The third French and second German editions were from the third English, with some few of the additions given in the fourth edition. A new fourth French edition has been translated by Colonel Moulinié; of which the first half is from the fifth English, and the latter half from the present edition. A third German edition, under the superintendence of Professor Victor Carus, was from the fourth English edition; a fifth is now preparing by the same author from the present volume. The second American edition was from the English second, with a few of the additions given in the third; and a third American edition has been printed from the fifth English edition. The Italian is from the third, the Dutch and three Russian editions from the second English edition, and the Swedish from the fifth English edition.

It is useful to distinguish between variants resulting from human modification that does not change the object's identity (e.g., a change to accommodate another operating system) and modification that does change identity (e.g., King James Version of the Bible), however there are no established terms to capture this distinction. In part, this may be due to most systems being operated by a single institution, such as a development organization operating a configuration management system, and hence the kind of cross-organization change of ownership that typically accompanies changes in identity cannot be captured. Literary Machines [137] uses the term *versioning by descent* when the document owner creates a variant, and the term *versioning by inclusion* when another user creates a variant with distinct identity. However, these terms are limiting, since branches of a version tree are used not just for representing variants, but also for capturing distinct revisions that are used to isolate the work of simultaneous collaborators. The term "versioning by inclusion" isn't precisely right either, since it is reasonable to discuss creating a variant by performing a copy operation, and hence the word "inclusion" ties the term to a particular implementation strategy. We prefer to use the term *variant* for non-identity changing alternates, and *alternate version* for externally derived variants that do change the object's identity (e.g., the Jimi Hendrix Star Spangled Banner is an alternate version of the American national anthem.)

Variation among compound, or aggregate, objects is also possible [123]. In Software Engineering, it is possible for variants of a software system to have different parts lists for each system variant. Alternate structures for the same set of objects are also possible [179], for example, different relationships among the parts of a software system.

Revisions, variants, and unversioned objects are all persistently stored instances of data (D) or processes (P_c) that provide input to a renderer that creates a symbolic rendition. The distinguishing factor between these items is their associated metadata, and whether they are kept, or overwritten, when modifications are persistently stored. Though a revision and a variant are distinguished by the kind of metadata associated with them, in practice they are not so distinct. Variants frequently record predecessor and successor information, and revisions can record information concerning points on axes of variation. The mechanisms used to record revisions and variants are often the same, differing mostly in the metadata they automatically store. As a result, we view revisions and variants as being, from a data modeling viewpoint, nearly identical. However, the kinds of operations performed on revisions, and on variants, do differ.

Revisions and variants are both denoted with a V, and in cases where they need to be distinguished, a revision is denoted V_{rev}, and a variant is V_{var}. Revisions of specific abstractions are denoted with a subscript, such as V_l for links, and V_a for anchors. The process of persistently recording a revision or variant from a data item or computational process is denoted P. Individual revisions and variants are denoted by subscripts, as in V_n ,$V_{rev,n}$, or $V_{var,\ n}$. Hence, the act of persistently recording a specific data item or computational process as a version or variant is expressed as:

$V_n = P(D(t))$ or

$V_n = P(P_c(t))$ for given t.

The notation $P_c(t)$ represents the process itself, not its output, at time t. Link revisions and variants are represented as:

$V_{l,n} = P(D_l(t))$ or

$V_{l,n} = P(P_{c,\ l}(t))$

Similarly, anchor revisions and variants are represented as:

$V_{a,n} = P(D_a(t))$ or

$V_{a,n} = P(P_{c,a}(t))$

2.3.3 Versioned Object

> *Versioned object.* An object that signifies a specific abstract work concept independent of any particular person, or instant in time. The versioned object contains revisions and variants associated with the abstract work concept.

As a signifier, the versioned object is a handle for an abstract work (or entity) concept. The versioned object does not represent the ideas that define the essence of the abstract work object, but is instead a handle for these ideas. Even though an abstract work concept varies across people and time, the association of a versioned object to that abstract work concept does not change. This is one reason why a versioned object is a handle: a handle does not need to represent the personal and time-varying differences in the abstract work concept, and hence is free to act as a neutral signifier for all authors and users of the work.

Since the distinguishing essence of an abstract work concept can encompass many possible revisions and variants, the versioned object has a containment relationship with revision and variant objects, meaning that symbolic representations of these objects evoke ideas within the observer that fall within the conceptual boundaries of the abstract work concept. Hence the versioned object has a dual role, as signifier for an abstract work concept, and as a container for revisions and variants.

A versioned object is denoted as *V-O*. Since many possible abstractions can be versioned, the subscript on *V-O* indicating what versioned abstraction the versioned object represents, with $V\text{-}O_w$ denoting a versioned work, $V\text{-}O_l$ a versioned link, and $V\text{-}O_a$ a versioned anchor.

Since a versioned object is a container, it can be modeled as a set of revisions or variants. The versioned object for a work is denoted:

$V\text{-}O_w = \{ V_i \mid \text{i} = 1 \text{ .. r, r is the number of revisions of } V\text{-}O_w \}$

Similar notation represents versioned objects for links and anchors:

$V\text{-}O_l = \{ V_{l,i} \mid \text{i} = 1 \text{ .. r, r is the number of revisions of } V\text{-}O_l \}$

$V\text{-}O_a = \{ V_{a,i} \mid \text{i} = 1 \text{ .. r, r is the number of revisions of } V\text{-}O_a \}$

A complete model of a work under version control is shown in Figure 5, which is Figure 1 updated with the addition of revisions/variants, and a versioned object.

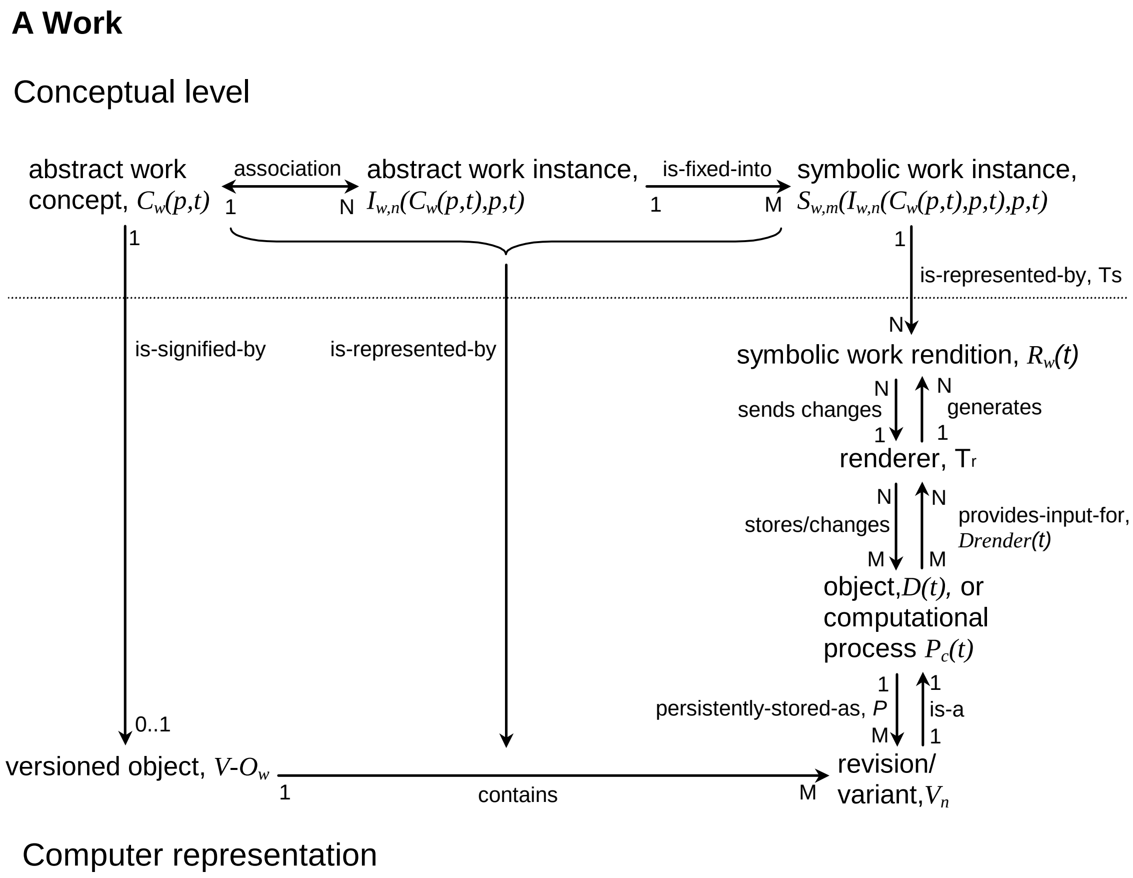

A Work
Conceptual level
abstract work concept, $C_w(p,t)$
association
1
N
abstract work instance, $I_{w,n}(C_w(p,t),p,t)$
is-fixed-into
1
M
symbolic work instance, $S_{w,m}(I_{w,n}(C_w(p,t),p,t),p,t)$
1
1
is-represented-by, Ts
N
is-signified-by
is-represented-by
symbolic work rendition, $R_w(t)$
N
N
sends changes
generates
1
1
renderer, T_r
N
N
stores/changes
provides-input-for, $Drender(t)$
M
M
object, $D(t)$, or computational process $P_c(t)$
1
1
persistently-stored-as, P
is-a
M
1
0..1
versioned object, $V\text{-}O_w$
1
contains
M
revision/ variant, V_n
Computer representation

Figure 5 – An intellectual work, with revisions and variants recorded, and contained within a versioned object.

Chapter 3

Taxonomy of Hypertext Versioning Systems

3.1 Introduction

This chapter provides a taxonomy of hypertext versioning systems, part of the characterization of the hypertext versioning domain. There are two broad categories in this taxonomy, divided into systems that address a particular problem or goal, as is the case with versioning for reference permanence, Web versioning, and versioning for open hypertext systems, and systems that have a particular data model, as with systems that have versioned data and unversioned structure, and composite-based systems. Due to this division into goal directed and data model oriented bins, the categories are not entirely orthogonal, and some systems fit into multiple categories.

3.2 Versioning for Reference Permanence

Ted Nelson, describing the Xanadu system in Literary Machines [137], was the first to fully embrace the problems inherent in changing documents and links, recognizing that change, like a cancer, slowly eats away at the consistency of relationship structures. If an entire document is deleted, links to it dangle. When a document is moved, links break unless repaired. The same problems recur inside documents when they're edited, where a linked-to region may be deleted, moved, rearranged, or otherwise modified, with links playing catch-up to maintain the consistency of the original relationship.

> "But *if you are going to have links you really need historical backtrack and alternative versions.* Why? Because if you make some links to another author's document on Monday and its author goes on making changes, perhaps on Wednesday you'd like to follow those links into the present version. They'd better still be attached to the right parts, even though the parts may have moved." [137, p. 2/25]

The Xanadu solution is to remember everything, prohibiting moves and deletes, while storing every change made to every document. In this scheme, links never dangle because linked documents are always present in their original form. While maintaining alternate document versions, and the intercomparison of alternates and versions is also important, it is maintaining the stability of references that most drives the design of Xanadu's version support. It is this goal that forces every change to be stored, and leads to move and delete being forbidden operations. This goal also introduces a decidedly hypertext issue into the realm of versioning, making it a valid topic of study in the hypertext community. Several researchers followed Nelson in the exploration of this topic, notably Vitali, Maoili, and Sola in the Rhythm system [124], Davis in his dissertation on link consistency in open hypertext systems [41], by Simonson, Berleant et alli in version augmenting URIs to achieve reference permanence on the Web [173], and by Durand in Palimpsest [56], and Vitali and Durand in VTML [194] (though Palimpsest and VTML were also strongly motivated by the goal of supporting collaborative authoring).

3.3 Versioned Data, Unversioned Structure

The hypertext systems KMS [4], DIF [78], Hyperform [202], and the Online Design Journal proposal [112], added version support, but only for objects. Versioning of links, and hence structure, is not supported, consistent with their (implicit) view that links are invariant, and hence do not require independent change tracking. Similarly, these systems have no configuration management support. Unlike Xanadu, where the emphasis was on persistently storing very fine-grain changes to preserve link consistency, these systems version content at the object level, and do not track fine grain changes. Thus, they persistently store important states of the objects, without recording the sequence of changes between steps. This makes it more difficult to preserve link integrity in general, and impossible in cases where a linked region has been extensively altered. Despite their similarities in version control support, these systems differ significantly in emphasis, with KMS focused on creating a full environment for hypertext authoring and browsing, DIF interested in hypertext support for software development environments, and Hyperform concerned primarily with separation of concerns in a hyperbase architecture. It is due to these diverse other interests that these systems do not explore hypertext versioning issues in depth.

3.4 Composite-based Systems

In the Neptune system [45,46], Norman Delisle and Mayer Schwartz grappled with hypertext versioning, but with a different motivation. Instead of preserving link consistency, or merely versioning individual objects, for them the key problem is collaborative teams writing hypertext documents. In order to collect related objects and links together, to provide isolated work areas for each collaborator, and to support selection of consistent groups of individual object and link versions, Neptune employs a key abstraction called a *composite*. (In fact, Neptune calls this abstraction a *context*, but we prefer the Dexter term, composite. Both can be used interchangeably.) A composite (context) is just a container object, and within a composite logically related items can be grouped, such as the sections of a paper, and links between those sections. When each collaborator works within a separate copy of the composite, it provides the appearance that each collaborator is working on their own individual paper. The drawback of this work isolation is that to create the final, complete paper, each individual's contributions must later be merged. Versioning is a key support technology for this, since it allows each collaborator's changes to be tracked, and merges to be recorded, while requiring that each composite select a consistent set of versions and links within which a collaborator works.

Composites collect together into one abstraction three concerns that are typically separated: collections or compound documents, typically used for collecting together related objects and links; *workspaces*, used to provide work isolation in many configuration management systems; and *configurations*, which provide the selection of consistent sets of objects and containment relationships. However, composites do address a uniquely hypertext problem, that of how to consistently version links between a consistent set of objects, and how to support evolution of this link structure, in conjunction with the evolution of the objects. The CoVer [87,89,88], VerSE [91], and HyperPro [141] systems, along with versioning support for the Nested Context Model [27] in HyperProp [179,178], and Melly's versioning support for Microcosm [128] all share Neptune's goal of exploring how to support hypertext structure versioning, and team document authoring, using composites.

3.4.1 PIE, a Change-Oriented Composite-Based System

Two landmark events for establishing the legitimacy of hypertext versioning as a research topic are Frank Halasz's 1988 Communications of the ACM article, "Reflections on Notecards, Seven Issues for the Next Generation of Hypermedia Systems," [96] (based on a presentation at Hypertext'87) and his Hypertext '91 conference keynote, "Seven Issues, Revisited," [97], since both identify versioning as one of seven critical issues for future hypertext systems. Version control was subsequently noted as an important issue for hypertext databases in the 1992 NSF Workshop on Hyperbase Systems [117], and the Hypertext'93 Workshop on Hyperbase Systems [115], cementing its importance within the community. The Halasz issues and the two workshops both legitimized and motivated hypertext versioning, making it easier to publish papers solely on this topic.

In addition to identifying versioning as a key issue, Halasz also singled out a configuration management system, PIE [80], asserting that, "the goal is to adopt and improve on the versioning mechanism that appeared in the PIE system" [97]. Though not originally designed as a hypertext system, PIE does allow arbitrary relationships to be defined between objects, and provides some support for navigating across these relationships, thus giving it a hypertext-like quality. PIE is the first change-oriented configuration management system [35], and Halasz extolled PIE's emphasis on logical changes that could span multiple objects and relationships, as opposed to versioning operations tailored to individual file operations (state-based). PIE stores a set of logical changes in a container object it calls a *layer*, which, because it can contain both objects and links, can also be viewed as a composite. But, because PIE only keeps changes in its composites, rather than a consistent state of the hypertext under development, and since these changes can be arbitrarily composed to create new hypertexts, PIE differs from Neptune [46], HyperPro [141], and HyperProp [178]. However, all these systems share the use of composites to contain the hyperdocument and its changes.

PIE directly influenced the work of Prevelakis [154], who set out to reimplement PIE specifically for hypertext application, independent of its original Smalltalk environment. Unfortunately, this work was never completed. PIE's change orientation also influenced Anja Haake's work on melding change-oriented and state-based versioning styles within CoVer [87], which has both composites that capture the entire state of the hypertext under development, as well as composites that capture only the changes between these states.

At the end of his 1991 keynote, Halasz reflected on the lack of versioning research since 1987, and noted that, “whether, in fact, versioning is important or not is still, I think, an important issue.” Ironically, at that time most work on the topic was just beginning.

3.5 Web Versioning

The Web’s lack of standard features for either browsing or authoring versioned content has led several researchers to investigate versioning for the Web. Hypertext links on the Web are embedded within HTML resources, and hence hypertext structure is not separate from data. As a result, it is not possible to version structure separate from data, leading to a focus just on versioning data (though Lo [120] does provide a proposal for separating unversioned links from embedded anchors in versioned SGML content).

Initial work concentrated only on browsing Web resources augmented with a version history. Typically, these systems append a version identifier to a URL, and augment a Web server to parse the version identifier, and retrieve the resource from a version store, such as an RCS [185] repository. Pettengill and Arango [150] adopt this approach to maintain different versions of materials in a digital library, as do Simonson and Berleant et alli [173], but to achieve reference permanence for all uses, not just digital libraries. Another common architecture for adding versioning services to the Web is the “form fill-in” style, exemplified by BSCW [19], WWRC [161], and V-Web [180]. These systems share the approach of using HTML pages to create a user interface for a revision control system, and work within the existing Web infrastructure to add versioning services. The limitations of this approach have led some to employ a “Java helper app.” approach, wherein a Java application is downloaded into the browser and acts as an intermediary between the remote versioned repository and the user’s local environment. Examples of this type of system are WWCM [104], MKS WebIntegrity [134], and WebRC [75]. Characteristic of all these approaches is their sole focus on versioning content, with no support for configuration management, or for versioning structure. Delta-V [199], a current working group within the Internet Engineering Task Force, is developing an application-layer network protocol for versioning and configuration management of Web content, but will not address full structure versioning since structure is embedded within HTML links.

Delta-V extends the WebDAV (Web-based Distributed Authoring and Versioning) protocol [201], itself an extension of HTTP [68].

The Web-based versioning and CM systems just described assume that the Web server is responsible for maintaining the predecessor and successor relationships between revisions. Research at NTT Laboratories resulted in a proposal [143] that shifts the relationship management to the client. In this approach, the HTTP LINK method (now deprecated) is used to create predecessor and successor relationships between resources in a version history, similar to [77]. This has the advantage that version histories can span multiple servers without requiring cooperation between these servers, but has the drawback that clients must be well behaved, as a single misbehaving client can corrupt a version history. A related approach is the non-Web-based NUCM [192] system, a client-server CM system in which a NUCM client interacts with a remote NUCM repository server. This interaction occurs using primitive operations (similar to those provided by HTTP [68] and WebDAV [68]) upon which are implemented higher-level CM styles. This is similar to the NTT Laboratories work in that the client is responsible for maintaining the consistency of the relationships in the remote repository.

3.6 Versioning for Open Hypertext

Several systems have been concerned with how to provide hypertext versioning support in an open hypermedia environment. The hypermedia version control framework developed by Hicks et al. [100] provides the HURL data model, along with a conceptual architecture that, together, are used by an open hyperbase to provide structure and data versioning services. Unique among the systems surveyed, this model supports computed anchors and links, and permits structure to be versioned independently from data. The data model and conceptual architecture were instantiated in the HB3 hyperbase management system [116], also described in [100]. Hyperform [202] similarly provides versioning services in a hyperbase, but with fewer services. The proposal for adding versioning to the Chimera system [200] shares the goal of providing structure versioning for open hypertext, but does so for an open linkbase system, where the data is controlled by an external repository. As a result, a focus of this work is how to associate and synchronize the versioned structure with the externally versioned data. Melly's work on versioning in Microcosm [128] also addresses hypertext versioning for an open linkbase system, but does so by creating context-like structures called *applications* that contain references to an active set of documents and links, using the context to avoid the document and linkbase synchronization issues in [200].

Chapter 4

Containment

4.1 Introduction

In our daily lives, we use containers all the time. Students use backpacks or bags to carry their books, and travelers use suitcases to carry their clothes. When shopping, we place our purchases into a basket or cart, and then carry home the goods in a shopping bag. If we drove to the store, these bags are placed in the trunk of our car, making nested containers: goods in bag in trunk in car. In all of these examples, the item is physically contained within the container, and can only belong to one container at a time.

A library is also a container, a building, which holds numerous books, maps, microfilms, videotapes, and other materials. However, a library's collection consists of more than just the materials physically present, since at any one time, many of these items have been checked out, and are in the possession of a single library patron. The library's catalog contains the complete list of a its collection, and refers to each of the items in its collection using a call number, such as those based on the U.S. Library of Congress notation [28]. This call number can be used to locate the item on a shelf, or in a special collection, and can be used by a librarian to determine which patron has checked out the book.

Unlike physical items, objects in a computer have the quality of easy duplication at low to trivial cost, and this means that computer containment is not zero-sum: the same object can belong to multiple containers. Consider a group of three pieces of (quite physical) fruit, an apple, an orange, and a banana. It is possible to give each piece of fruit a reference number, such as fruit1, fruit2, and fruit3, and then create two collections of the same fruit, each containing the three fruit by writing the reference numbers on a sheet of paper. This system works well until someone actually wants to use one of the pieces of fruit. Even the nondestructive act of examining, say, the apple, requires replacing the reference number (fruit1) with the apple, thus denying access to the apple by the other collection, even though it holds a reference to fruit1. Computers don't have this problem. Since objects can be quickly and cheaply duplicated, when one collection wants to read an object, they're given an in-memory copy. Then, if another collection also wants to read the same object, it too is given a copy. Of course, modification and destruction of a computer object still makes it unavailable to all collections, just like they do for physical objects. But, the ease of object duplication afforded by computers dramatically increases the utility of containing objects using references, and holding the same object in multiple containers.

Computer containers fill many roles, providing organization of large collections of objects into smaller units, a form of modularization (exemplified in the hypertext versioning literature by [178,46,141,100]), and information hiding via encapsulation [91,178]. Containers can also be used to model compound documents, for example, the combination of some text and image objects to model a document containing figures; Dexter composites exemplify this use [95]. Hypertext links are a form of container, as described in Section 2.2.3. Just as with physical containers, computer containers are used to transport items, examples including ZIP files, Internet Protocol (IP) packets, and the MIME multipart/related packaging of documents in electronic mail [118].

This chapter describes the various forms of computer containment that appear in hypertext, document management, and configuration management systems. It begins with an examination of basic static containment, where the intent is to model a pure set, with no constraints on the number, or type of objects that can be included, and included objects are explicitly listed. The common terms of inclusion and reference containment are next defined using the basic static containment framework. Augmenting the capabilities of basic static containment, some advanced containment functionality is outlined, such as allowing the type of included objects to be constrained, and the internal structure to be specified. Dynamic containment, which allows included objects to be determined by a computational process, is described at the end of the section.

4.2 Basic Static Containment

In its basic form, a container models a set where each element is an entity (an abstraction). The container is an entity that holds the set. The *containment relationship set* is a mathematical relation among container and containee entities:

$\{[c, e] \mid c \in \text{Containers}, e \in \text{Containees}\}$

where each pair of entities [c, e] is a *containment relationship* between one element of the set of containers, c, and one element of the set of containees, e. The predicate contains(c, e) is true when e is a member of c, or contains(c, e) = $c \in$ Containers ^ $e \in$ Containees. The *containment set*, C_{set}, is the set of entities actually held within a given container, c:

$\{[e_1, e_2, \ldots, e_n] \mid e_n \in \text{Containees} \wedge \text{contains}(c, e_n)\}$

There are two main aspects to the containment design space:

- **Abstract properties of the container:** Qualities of the container that are mathematic set properties, rather than properties of a specific computer representation, these being:
    - *Containment*: For a given entity, the number of containers that can hold it. Choices are: (a) single containment, an entity belongs to just one containment set, or (b) multiple containment, an entity belongs to multiple containment sets,
    - *Membership*: For a given container, the number of times can it contain a given entity. Choices are: (a) single membership, an entity can belong to a containment set only once, or (b) multiple membership, an entity can belong to a containment set multiple times, in which case the containment set is a bag, or multiset,
    - *Ordering*: The persistent ordering of a container. Choices are: (a) ordered, the entities within the containment set have a fixed successive arrangement, or (b) unordered, the entities have no prescribed arrangement;
- **Containment type:** How containment relationships are represented.

Broadly, there are two ways to represent that a container contains a particular entity. The container can physically include the contained item, or it can use an identifier as a reference to its members. The former case is known as inclusion containment; the latter, referential. Whenever two entities have a relationship between them, this relationship can be represented using references, following the permutations of identifier storage. The identifier can be stored on the container, on the containee, or on both. Additionally, identifier storage can be delegated to a separate entity, a first-class relationship. However, since the relationship is itself a container, the same permutations of identifier storage apply between the container and one endpoint of the relationship, and between the containee and the other endpoint. Typically the relationship holds identifiers for both the container and containee.

One consequence of using containment relationships of any kind is the possibility of containment relationship cycles. Containment cycles occur when a container contains itself, either directly or indirectly, via one of its contained containers. Systems vary on whether they allow containment cycles. On the one hand, it is typically inefficient to detect cycles when they are created, thus leading to systems permitting cycles. On the other hand, cycles add complexity to operations on containment graphs, since the cycles must be detected to avoid infinite processing. This leads to the desire to prevent the creation of cycles.

The primary containment types are detailed below.

- **Inclusion:** Members are physically included within the container, as shown in Figure 6(b) below. Inclusion containment is frequently called aggregation [23,164], since the container aggregates, or combines together, the contained entities. For this reason, it is also described as representing a *part-of* relationship, since the contained entity is a part of the container.

  An unusual type of physical inclusion occurs when the container encapsulates the type definition of contained entities, in addition to containing its state. In this case it is not possible even to copy such a contained entity, since the type definition is not externally visible. By comparison, physically included entities can typically be copied to another container, since the container does not hold their type definition. One example of type definition encapsulation occurs in structured program editors, where definitions of loop constructs and conditionals depend on the program block that contains them [86], p. 86.

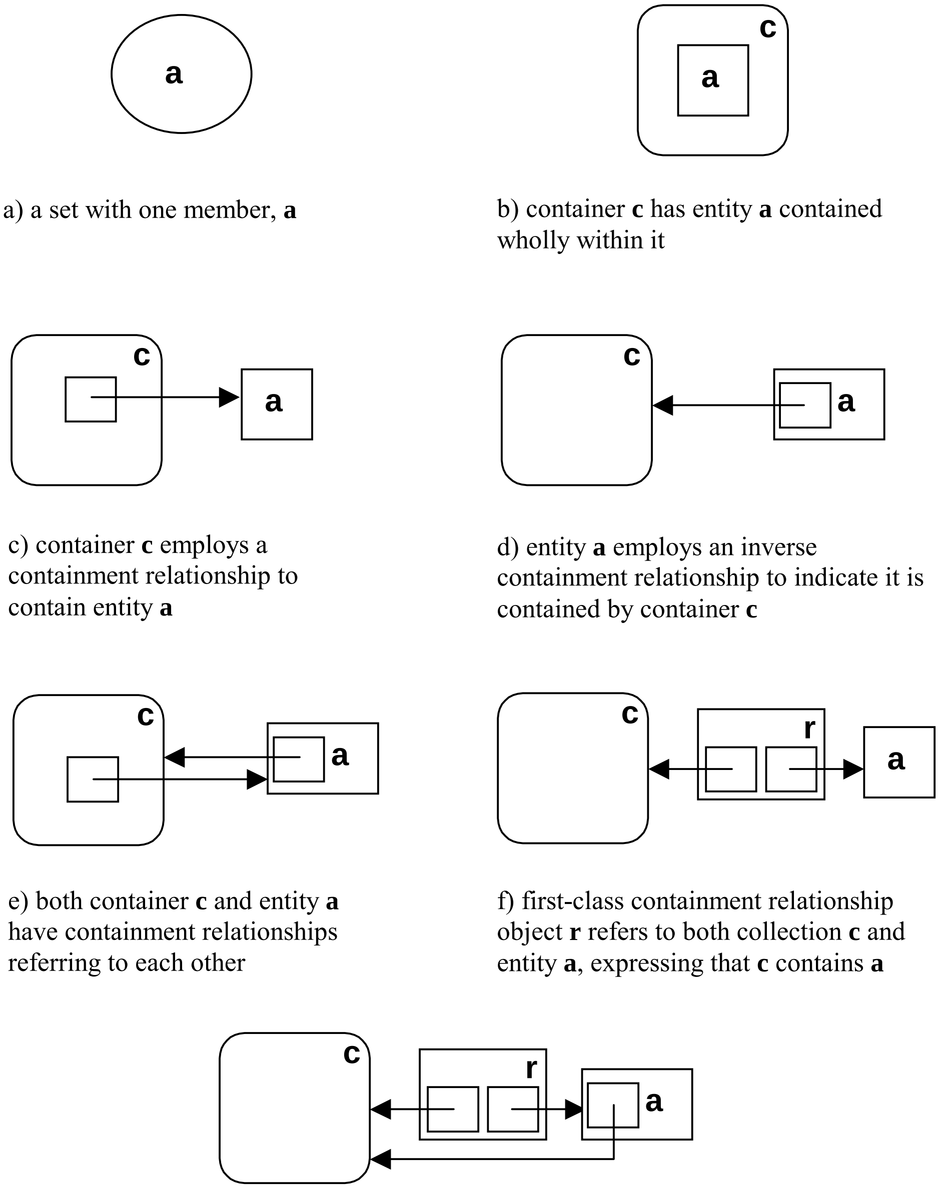
The following containment types are used for referential containment:

- **Containment relationship on container:** The container provides storage for the identifier of the containee, and thus owns the containment relationship. Containees do not know which collections contain them, or even if they are contained at all. This provides the main advantage for this approach: a container can add any entity as a member without modifying it, independent of whether it is writeable or read-only. This is shown in Figure 6(c) below.

In object organizations where properties are present (data plus properties, or all properties), it is possible for the containment relationships to be stored within one or more properties. When an object has multiple properties, each holding a set of containment relationships, a single object can represent many containers. The DeltaV protocol [199] is an example of this, where each history resource has both a property containing a list of revisions, and a property containing a list of working resources.

- **Containment relationship on containee:** The containee owns and provides storage for the identifier, and owns the containment relationship. The container does not know what entities it contains, or if it contains any objects at all. This is useful when users have access to contained items, but not to collections. Despite not knowing its members, the container can still be used to hold metadata about the collection. This is shown in Figure 6(d) below.
- **Containment relationship on container and containee:** Both the container and its containees own and provide storage for containment relationships (the contained entities have the inverse containment relationship). That is, for each containee, there is a containment relationship on the container, as well as one on the containee, often called a backpointer. In this case, the container knows what entities it contains, and the containees know what containers contain them. This is shown in Figure 6(e) below.
- **First class containment relationship:** An entity that is neither the container, nor the containee, records the containment relationship. With no further information, the container does not know what entities it contains, and the containees do not know what containers contain them. This is shown in Figure 6(f) below. However, if pointers on both the container and the containee are added to refer to the containment relationship object, then it is possible for the container to know its contents, and vice-versa. A significant advantage of this approach is that it permits the storage of metadata on the containment relationship, such as who added a containee, and when. This metadata is not handled well when the containment relationship is part of the container or containee, since there is no entity to which the metadata can be attached. The Document Management Alliance (DMA) 1.0 specification [50] uses first class containment relationships for this reason.

- **Hybrid:** Combinations of these basic containment types are also possible. For example, the DMA 1.0 specification [50] combines its first class containment relationships with a relationship on containees that points back to its parent container, as shown in Figure 6(g). Note that DMA only uses this backpointer when the containee belongs to just a single container (what the DMA 1.0 specification calls direct containment). This additional relationship allows for more efficient navigation from containee to parent container.



a) a set with one member, **a**

b) container **c** has entity **a** contained wholly within it

c) container **c** employs a containment relationship to contain entity **a**

d) entity **a** employs an inverse containment relationship to indicate it is contained by container **c**

e) both container **c** and entity **a** have containment relationships referring to each other

f) first-class containment relationship object **r** refers to both collection **c** and entity **a**, expressing that **c** contains **a**

g) a hybrid containment structure, where a first-class containment relationship **r** refers to both collection **c** and entity **a**, and **a** also has a containment relationship pointing back at **c**, that bypasses **r**.

Figure 6 – The five primary types of containment, plus one hybrid containment type.

Highlighting the descriptive range of this containment model, Table 2 below details the containment type employed by the container object (composite, context, etc.) within selected hypertext versioning systems, along with the Dexter hypertext model [95], and the DMA 1.0 standard [50]. The containment characteristics for links, which are also a type of container, are not shown in this table.

System & Object	**Single/Multiple Containment**	**Single/Multiple Membership**	**Ordered / Unordered**	**Containment Type**	**Cycles Permitted**
Neptune [46] *context*	Single	Single	Unordered	Container holds relationship	No
Dexter [95] *composite*	Multiple	Single	Unordered	Container holds relationship	No
HyperProp (Nested Context Model) [179] *composite*	Multiple	Multiple	Ordered	Container holds relationship	No
HyperPro [141] *composite*	Multiple	Multiple	Ordered	Container holds relationship	No
CoVer [87] *composite*	Multiple	Multiple	Partially ordered	Container holds relationship	Yes
DMA 1.0 [50] *container*	Both	Single	Optional	Hybrid. First-class containment relationship, with additional backpointer on object pointing to container.	Yes
Hypermedia Version Control Framework [100] *composite*	Multiple	Single	Unordered	Container holds relationship	Yes

Table 2 – Static containment characteristics for composite-based hypertext versioning systems, the Dexter [95] reference architecture, and the DMA 1.0 standard [50].

4.2.1 Independence of Mathematic Set Properties

Together, the mathematic set properties, the containment type, and whether cycles are permitted, define the axes of the design space for static containment. One concern is the independence of these axes. Does a design choice on one axis restrict or imply choices on another? The mathematic set properties are completely independent of each other. Whether an object can belong to only a single container, or to multiple containers does not affect whether that object can appear once, or more than once in a single container. Similarly, the existence, or lack of an ordering on a collection does not affect the set's containment or membership properties. However, there are some interactions between the containment type, and the set properties, as shown in Table 3. Containment by inclusion does preclude having an object participate in more than one container, and makes containment cycles impossible. Additionally, inclusion containment makes it more difficult to have the same object appear multiple times in the same container. However, containment via containment relationship does not introduce any additional dependencies between design axes.

	Object can only belong to one container	**Object can belong to multiple containers**	**Object can appear only once per container**	**Object can appear multiple times per container**	**Ordered container**	**Unordered container**
Inclusion	Yes	No	Yes	Possible by duplicating object within the container	Yes	Yes
Container holds containment relationship	Yes	Yes	Yes	Yes	Yes	Yes
Object holds containment relationship	Yes	Yes	Yes	Yes	Possible by having object record its order within each collection	Yes
Containment relationship on container and object	Yes	Yes	Yes	Yes	Yes	Yes
Independent containment relationship object	Yes	Yes	Yes	Yes	Yes	Yes

Table 3 – Interactions between containment type, and mathematic set properties of containers. A "yes" entry indicates that the containment type is capable of representing the set property.

4.2.2 Deletion Semantics

Since containment involves two objects, the container and the containee, deletion of either of these objects affects the containment. There are three deletion operations of interest: delete a container object, delete(container), delete a contained object, delete(object), and delete an object from a specific container, delete(container, object). Table 4 below enumerates how the semantics of these delete operations vary with containment type.

	delete(container)	**delete(object)**	**delete(container, object)**
Inclusion	Causes deletion of contained objects too.	Causes removal of object from container.	Causes removal of object from container
Container holds containment relationship	Container is deleted, along with all containment relationships, and either: a) all contained objects are left untouched, or, b) all contained objects are deleted as well, causing recursive deletion of member containers (same semantics as inclusion containment)	Object is deleted, and either: a) all containment relationships in all containers that refer to the object are deleted, or, b) no relationship cleanup is performed, causing containment relationships that contain the object to dangle.	Removes containment relationship and then either: a) leave object untouched, or, b) delete contained object (same delete semantics as inclusion containment), or, c) delete object only if it is no longer being contained by any collection (i.e., delete with garbage collection).
Object holds containment relationship	Container is deleted, and either: a) remove all containment relationships in all objects that refer to the container, or, b) no relationship cleanup is performed, causing containment relationships that contain the container to dangle.	The object, and all its containment relationships are deleted.	Either: a) only the containment relationship referring to the specified container is removed, or, b) the object and all its containment relationships are deleted.
Containment relationship on container and object	Container is deleted, and either: a) all containment relationships in all objects that refer to the container are deleted, or, b) no relationship cleanup is performed, causing containment relationships that contain the container to dangle (same as for when the object holds the containment relationship).	Object is deleted, and either: a) all containment relationships in all containers that refer to the object are deleted, or, b) no relationship cleanup is performed, causing containment relationships that contain the object to dangle (same as for when the container holds the containment relationship).	Either: a) remove both the containment relationship on the container and on the object, and leave the object untouched, or, b) remove both containment relationships, and remove the object only if it is no longer contained by any container (garbage collection), or, c) remove the containment relationship on the specified container, then perform a delete(object).

Independent containment relationship object	Container is deleted, and either: a) all containment relationships that refer to the container are deleted, or, b) no relationship cleanup is performed, causing containment relationships that contain the container to have one invalid endpoint.	Object is deleted, and either: a) all containment relationships that refer to the object are deleted, or, b) no relationship cleanup is performed, causing containment relationships that contain the object to have one invalid endpoint.	Either: a) remove the containment relationship between the object and the container, leaving the container and object untouched, or, b) remove the containment relationship, and remove the object only if it is no longer referenced by any containment relationship (garbage collection), or, c) remove the containment relationship and perform a delete(object).

Table 4 – Variations in delete semantics for different containment types.

4.3 Relationship Abstraction Layers

At its most abstract, a container has an undifferentiated *contains* relationship between itself and its containees. This contains relationship is refined by specifying the abstract containment set properties of containment, membership, and ordering, along with the containment type. There are many permutations of the abstract containment set properties and containment types, and hence there are many possible ways to refine a contains relationship into specific container characteristics.

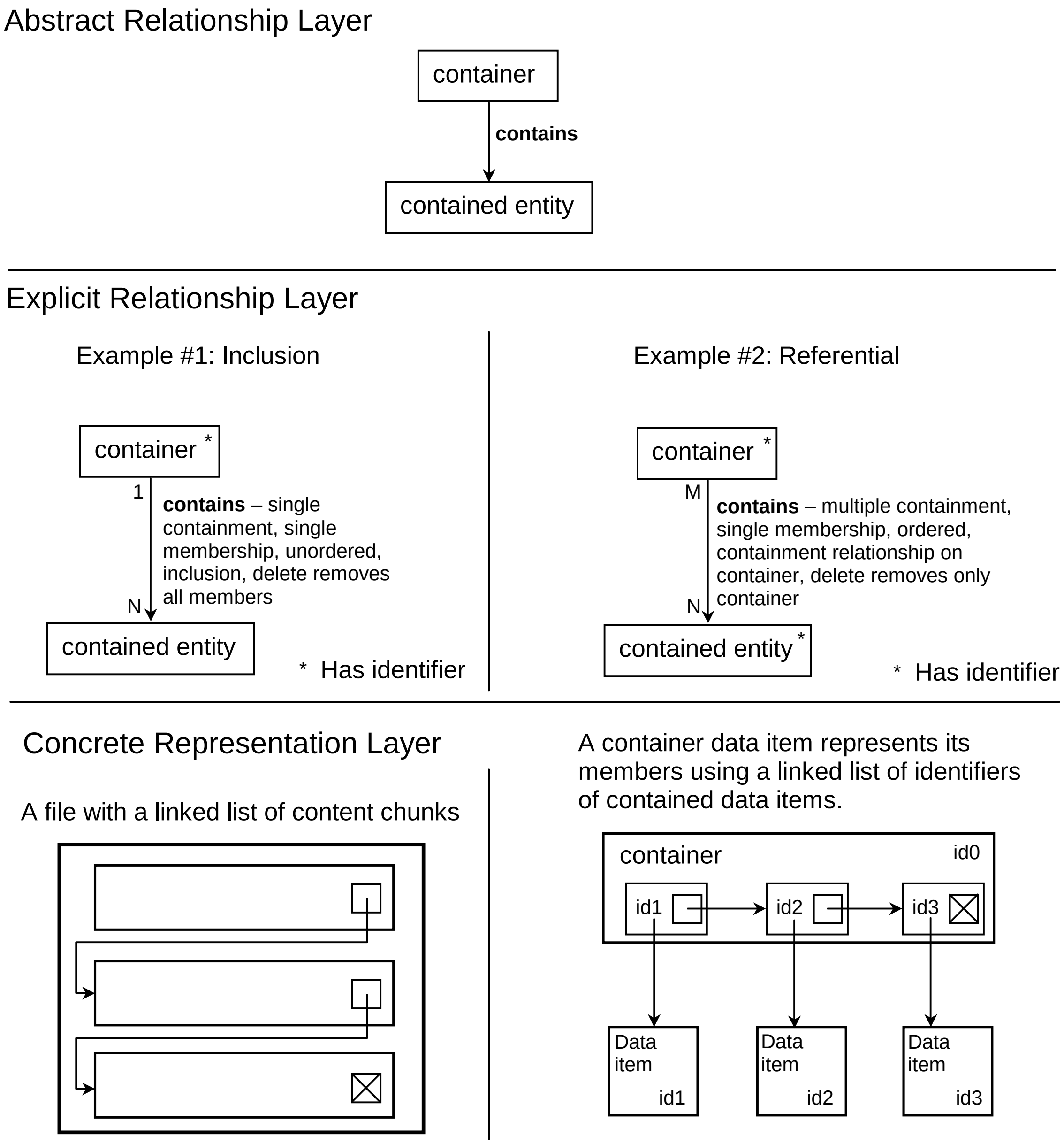
Similarly, there are many possible computer data structures that can be used to represent a specific container as concrete data items [2]. Examples include arrays, linked lists, hashed lists, comma-separated text strings, and various types of trees, to name just a few. These data structures all support the operations of creating a set (or bag—all the following operations apply to bags too), inserting a member in a set, listing the members of a set, deleting a member from a set, and deleting a set. Ordered sets add position information to the insert operation, and additionally add an operation to order some members of the set. Other set operations are also possible, such as union, intersection, difference, etc., but are less frequently used by containers.

A container's concrete representation can be viewed as an abstraction layer that implements a specific containment relationship that has had its containment, membership, ordering, and containment type precisely specified. This specified containment relationship is also in an abstraction layer that refines a more abstract model of containment in which there is just an undifferentiated contains relationship between containers and their contained entities.

The abstraction layer holding the undifferentiated contains relationship is termed the *abstract relationship layer*, since it provides an abstract depiction of the contains relationship, providing only the type of the relationships, and omitting all other details concerning its specific properties. Entities in this layer are abstraction signifiers, indicating that they have distinct intellectual identity as abstractions, irrespective of whether their eventual concrete representation has independent identity. The abstract relationship layer is shown at the top of Figure 7.

Precisely specifying the characteristics of the relationships in the abstract relationship layer results in a more detailed depiction in the *explicit relationship layer*. Containment relationships at this layer have fully specified their containment, membership, and ordering properties. Containers at this layer describe whether they are using inclusion or referential containment, and, if referential, the details of where identifiers are kept. Similar to the abstract relationship layer, entities in the explicit relationship layer are abstraction signifiers. No refinement of entities occurs between the abstract and explicit relationship layers. However, since containment relationships are differentiated between inclusion and referential containment, entities indicate whether they have an associated identifier. The explicit relationship layer is shown in the middle of Figure 7, which depicts two possible ways to refine the contains relationship in the abstract relationship layer into explicitly defined containment relationships. If an entity has an associated identifier, an asterisk represents this.

As noted above, there are many possible ways to map the entities and relationships of the explicit relationship layer into concrete data items within a computer. In the *concrete representation layer*, abstract entities and relationships have been reified into specific data structures and chunks of state. Repositories such as databases and filesystems can be used to realize the concrete representation. These systems themselves are complex, and often have several layers of abstraction within their implementation. However, this containment model does not model these additional implementation abstraction layers. Figure 7 shows one possible concrete representation, out of the universe of possible representations, for each of the examples in the explicit relationship layer. The inclusion containment example is reified as a file that internally has a linked list of data chunks, which are each a sequence of bytes. The referential containment example is represented using a container data item that holds within it a linked list of identifiers to contained data items. The internal structure of contained data items is unconstrained. Both the container and containee data items have identifiers.



Abstract Relationship Layer
container
contains
contained entity
Explicit Relationship Layer
Example #1: Inclusion
container *
1
contains – single containment, single membership, unordered, inclusion, delete removes all members
N
contained entity
* Has identifier
Example #2: Referential
container *
M
contains – multiple containment, single membership, ordered, containment relationship on container, delete removes only container
N
contained entity *
* Has identifier
Concrete Representation Layer
A file with a linked list of content chunks
A container data item represents its members using a linked list of identifiers of contained data items.
container
id0
id1
id2
id3
Data item id1
Data item id2
Data item id3

Figure 7 – A container at three different layers of abstraction. An abstract containment relation (abstract relationship layer) is refined into one example each of inclusion and referential containment. The explicit relationship layer fully details the containment relation, specifying containment, membership, ordering, and containment type. For the inclusion and referential examples, the concrete representation layer shows an example reification of the containers as persistent data items.

4.4 Common Definitions of Inclusion and Reference Containment

Having enumerated the mathematic set properties, the containment types, and the delete semantics, it is possible to relate the commonly used terms *inclusion containment* (or containment *by-value*) and *reference containment* to this model. Since the exact meaning of inclusion containment varies with use, this text uses the most common hypertext definition, as given in [96,86], of a "part/whole relationship in which characteristics of and operations on the whole will affect the parts as well," [96], p.844. There are several points in the containment design space that can satisfy this definition. Certainly physical inclusion, or aggregation, where the contained object is stored as part of the container, meets the definition. However, any kind of referential containment can also satisfy this definition so long as the object is contained in only one container, and deletion of the container implies deletion of all contained objects. Note that this is true even when the object holds the containment relationship, assuming it is possible to determine all the objects belonging to a container, for example by using a search facility across all system objects looking for containment relationship values that identify the container. As a result, the common definition of inclusion containment *does not constrain the containment type at all*, instead limiting each object to one container, with deletion of a container implying deletion of all members.

In contrast, there is reference containment, defined as "a much looser relationship, in which the participating entities allude to each other but remain essentially independent" [96], p. 844, or the more precise "allow for the same component to be a member of several {containers}," [86], p. 85. These definitions preclude using physical inclusion, but otherwise are compatible with all containment types that employ containment relationships. Objects can be a member of one, or more than one container. The biggest difference between reference containment and inclusion containment is in deletion semantics, since reference containment requires that deletion of a container must not result in deletion of its members. Table 5 below summarizes the differences between the common definitions of inclusion containment and reference containment.

Inclusion Containment	**Reference Containment**
• All containment types	• All containment types except physical inclusion
• Objects can belong to just one container	• Objects can belong to one, or many containers
• Deletion of container causes deletion of contained objects	• Deletion of container does not result in deletion of contained objects
• Cycles not permitted	• Cycles are possible, though may not be permitted

Does not constrain:

- Whether an object can appear once, or many times in the container
- Whether a collection is ordered, or unordered

Table 5 – Characteristics of inclusion and reference containment, according to common definitions of these terms as generally used in hypertext systems. Note that in this document, inclusion containment strictly means physical inclusion (the definition given in Section 4.2), and does not use the definition in this table.

4.5 Links, Containment Relationships, and Containers

The existence of first-class relationships as a type of referential containment begs the question of the differences and similarities between hypertext links and containment relationships. From a data modeling perspective, links and containment relationships are very similar: containment relationships on containers or objects are a kind of unidirectional embedded link, and first-class containment relationships are a kind of bi-directional link. In both these cases, the link is to an entire object, not to an anchor point within the object, and so this highlights one difference between links and containment relationships: containment relationships are constrained to have entire objects as their endpoints. Still, some hypertext systems use hypertext links to implement containment relationships, exploiting their inherent similarity.

However, though containment relationships are very similar to hypertext links, there is a big difference between the semantics associated with a container object, and a generic (non-container) object type. While generic objects can be made to act somewhat like container objects by linking them to other generic objects (perhaps even by using a special containment link type), the system does not know that the user intends and expects a particular generic object to behave like a container object. Inclusion deletion semantics are one example [96]. Using just links between generic objects, a hypertext system will not be able to provide by-value delete semantics. Instead of the desired outcome, where the collection and all of its members are removed in one operation, the system will do the only thing it knows how to do, deleting just the one generic object, and perhaps performing link cleanup.

Similarly, operations that require understanding about containment, such as those that operate on compound documents, will behave unexpectedly when interacting against a container implemented with generic objects. For example, Halasz, in his critique of NoteCards, notes that while it is possible to use link structures to hierarchically organize documents into sections, NoteCards' lack of knowledge about containment reduces its ability to operate on documents as a whole. While a utility program can reconstruct a hierarchically organized document into a single card holding the entire document contents, subsequent modifications to the document card do not automatically cause the appropriate sub-section card to be updated, and vice-versa. If NoteCards understood that the specific object was being used as a compound document, it would be able to provide better support for this scenario [96].

So, in summary, while containment relationships and hypertext links are very similar, containers are distinct from other non-container objects due to their container-specific knowledge and behavior.

4.6 Dynamic Containment

For all containment types except for inclusion, a container can be viewed as a mapping from the set of all objects to the set of all containers. For single containment, this mapping is M:1, where M is the number of objects, while for multiple containment, this mapping is M:N, where N is the number of collections. With static containment, the set of members is explicitly listed. For dynamic containment, the mapping from objects to collections is generated by a function.

Queries are by far the most common functions used to dynamically create containers. DHM [84], the Hypermedia Version Control Framework [100], and CoVer [88] are all examples of systems that support the population of containers from query results. There are two ways dynamic containment can be employed:

- **Query results specify the endpoint of one containment relationship:** In this approach, each containment relationship can have a query associated with it, and this query typically is designed to return a single result, the endpoint. Thus the query determines a single element of the container. This approach is frequently employed to pick out a single revision from all the revisions of an object by scoping the query to just a single versioned object, and by selecting a query predicate that returns just a single revision. When scoped to a single revision history, the query predicate is termed a *revision selection rule*. The Hypermedia Version Control Framework supports arbitrary queries for endpoint selection.

- **Query results specify the membership of a collection:** Here the results of a query comprise the entire contents of the collection. In DHM, one system that supports this kind of containment, these containers are called virtual composites.

4.7 Advanced Containment Semantics

Some containers offer additional capabilities that extend the containment design space. These are:

- **Type of contained objects:** Especially in systems that support a wide variety of object types, containers may limit, or explicitly state the type of objects that can be contained. For example, in the Hypermedia Version Control Framework [100], the association set is a container that can only contain associations, and in Aquanet [126], schema relations are containers that may restrict the type of contained objects. This issue is noted as the "Type" aspect of composite design in [86], p. 84.
- **Number of contained objects:** Containers may have a fixed size, or an upper bound on their size. For example, Aquanet schema relations can have a fixed number of contained items, such as when modeling an argument relation (see Figure 2 of [126]), which has two slots for statements, and one slot for rationale.
- **Typing of the container:** the previous two capabilities, specifying the number and type of contained objects, can be viewed as two kinds of constraints that would be given in the definition of a specific container type. Aquanet schemas essentially define new container types with each new relation, and DHM [84] provides several container subclasses, such as the GuidedTourComposite and TableTopComposite. Specific container types can express a wide range of structures, as noted in [86], p. 84-85. Compound documents can also be expressed using a container type that provides viewing and editing semantics for the contained objects that allows the container to behave like a single document, instead of a set of independent objects.

Chapter 5

Address and Name Spaces

All hypertext, document management, and configuration management systems employ names, addresses, or both, to identify objects within the system. These identifiers are required for referential containment, since they are used by the endpoint specification(s) within a containment relationship. A *name* is a human-understandable identifier for an object that may also be machine readable, while an *address* is solely a machine-readable identifier for an object that is not human-understandable, except by experts in the system. For example, within the Xanadu system [137], addresses called "tumblers" are used, and the Document Management Alliance 1.0 [50] specification employs addresses called Object Instance Identifiers (OIIDs), to identify persistent objects. The http URLs employed on the Web can serve as both names and addresses – the URL http://www.{company name}.com/ is commonly understood to be the name of a firm's Web page, yet there are many URLs that are not meaningful to a human, or actually have addresses embedded within them, and thus have the qualities of an address.

Starting with basic physical memory, be it disk, RAM, ROM, or other physical device, every chunk of memory has an address. Disks are divided into cylinders, tracks, and sectors, while RAM and ROM have memory addresses for each byte or word. Each abstraction layer built on top of these basic physical memories uses new sets of identifiers, with meaning specific to the abstraction. For example, the Unix inode is the basic identifier of a file, which in turn can be converted into a series of blocks, and each block in turn can be translated into a physical address on a disk. The *resolver* is the function that converts the identifiers at one level of abstraction into identifiers for the next abstraction layer down. A *redirector* function is sometimes available as well, performing a mapping of identifiers at the *same* level of abstraction. A redirected identifier still must be applied as input to the resolver function to access the identified object. The symbolic link functionality in Unix is an example of namespace redirection. Even after dereferencing a symbolic link into another filename, it still must be resolved into an inode before the actual file contents can be accessed.

The set of all possible addresses or names in a particular addressing or naming scheme is called either the *address space* or the *name space* for that scheme. Often the set of possible names is infinite.

The complete definition of a scheme for naming or addressing consists of:

- **Syntax:** the basic syntax of the names, or addresses. The quality of human readability is determined by syntax.
- **Semantics:** the meaning, if any, of subparts of the name or address. Names and addresses with no internally encoded semantics are termed *opaque*. Name or address semantics encompass uniqueness properties, as well as interactions with containment hierarchies, such as the interaction of filenames with directory structures.
- **Resolver:** the resolver function for the names, or addresses, specifying which names or addresses form the output of the resolver, the process used to resolve the name or address, and the system entity (architectural component) or entities that are engaged in the operation of the function.
- **Assigner:** when an object is created, what system entity, or entities, assigns the new name of the object. Note that not all name or address spaces have assigner functions – for example, a memory location in a chip is a physical property of the device itself, not something that is assigned to it by a system element. However, in some cases it is meaningful to describe which system entity assigns names of addresses. In the case of the mapping of URLs to Web resources, the origin Web server is responsible for name assignment.
- **Redirector:** if a redirector function is present, the semantics of the redirector function, specifying which names or addresses it operates on.

Hierarchical containment structures are frequently reflected in object names. As exemplified by a filename, the prevalent way names reflect containment is for the name of the parent containers to be concatenated together, each name separated by an character, often "/" or "\". Finally, to the name of its parent container, the individual file appends its own name. In this case, the semantics of names are completely intertwined with the semantics of containment. For example, if a file can be contained by multiple directories, then the file can have many names, yet if only single containment is allowed, then a file has just a single name.

Versioning too often affects names, by giving each revision of an object a version identifier, such as 1.2.1, which is appended to the name of the versioned object, as in “main.c, 1.2.1”. In essence, this creates a dual naming system, with one set of names used to identify specific versioned objects, and the second set identifying specific revisions of those objects. Since many versioning systems represent revisions as being contained by their versioned object, this is another example of containment affecting naming. Vitali and Durand provide a detailed description of three revision addressing schemes, outline numbering, reverse outline numbering, and L-shaped numbering, in [194].

One challenge emerges when hierarchical containment names and versioning containment names are combined. It is tempting to just add the revision identifier to the end of the name determined by the hierarchical containment structure. However, if containers are versioned as well, then associated with each container in the containment hierarchy are really two containers: one holding the objects currently contained by that revision of the container, and another holding all revisions of the container (the versioned object for the container). At each level in the containment hierarchy, it is now necessary to identify an element from each of these containers, yet most hierarchical naming schemes assume that each level in the containment hierarchy is associated with just a single name, located between separator characters (e.g., “/{name}/”) The challenge is how to shoehorn in the additional versioning information. Two approaches are to add the version identifier in at each level (e.g., “/{name},{version id}/”), or to have some external specification of each revision for each level.

5.1 Centralized Assignment and Resolution

A common simple addressing scheme is to assign each system object an address that is unique to that instance of the system. The system is responsible for assigning addresses when objects are created, and provides functions for resolving those addresses into objects by means of converting each address into internal system identifiers that can be used to retrieve the object. The address space is totally centralized, as names are assigned and resolved by the system.

The Chimera 1 system [7] is an example of a centrally assigned and resolved address space. All Chimera objects are assigned a long integer address when they are created, and this address uniquely identifies the object in the instantiation of the Chimera system in which it was created. All objects (e.g., anchors, links, views, etc.) share the same address space. While this addressing system is well suited for Chimera 1, it does not provide any support for distribution of objects across multiple instances of the Chimera system, or any support for interchange of hypertexts.

The Dexter [95] reference model also assigns each system object an address, but one that is unique across all instances of Dexter-compliant systems. Dexter addresses are called UIDs, for unique identifiers, and UIDs are assigned by an instance of the Dexter system when objects are created. Dexter also provides support for component specifications, which indirectly identify a specific object, such as with a simple search specification. In Dexter terminology, the function that converts a component specification into a UID is a resolver function, while the function that converts a UID into a specific object instance is called an accessor function.

Neptune [46], in its object management layer, the Hypertext Abstract Machine (HAM) [45], also provides unique identifiers for all system objects. The HAM is responsible for assigning and resolving these identifiers. Neptune requires each system object to be contained within one, and only one, container, known as a context. When a new context is derived from an old one, all objects in the old context are copied into the new one, in order to maintain single containment. This makes it possible to include the context identifier in each object’s address. So, Neptune object identifiers are an identifier pair consisting of an invariant identifier that is the same for all instances of the object, and a context identifier that acts as an instance identifier, since each object can belong to only one context. The benefit of this approach is that it permits the retrieval of all revisions of an object without needing a separate object to record the revision history. However, it does require integrating the time of last modification with the revision histories of the containing contexts to recreate the revision history of an individual object.

5.2 Decentralized Assignment and Resolution

Decentralized addressing and naming schemes provide identifiers for large pools of documents that are controlled by many different organizations. These decentralized identifier spaces allow the assigner and resolver functions to be replicated throughout the system, with each organization capable of controlling and delegating these functions. Hauzeur also notes that decentralized identifier resolution may involve routing services [98].

The Xanadu system [137], with its tumblers, provides an example of a sophisticated decentralized address space. Xanadu does not have any facility for naming objects. At their top level, tumblers are a series of fields, beginning with a "1.", and separated by ".0." as follows:

1.{node}.0.{user}.0.{document}.0.{element}

Each field is itself a sequence of dot-separated numbers with no fixed size, which allows each field to address an unbounded number of items. The node field contains the identifier for a specific Xanadu server (node), while the user field contains account, and potentially sub-account identification. The document field identifies the document, and potentially the version, and subdocument parts as well. Finally, the element field allows the specification of within-document parts.

A specific Xanadu system node assigns tumbler addresses when the canonical instance of an object is created. Resolution of a Xanadu tumbler is a two-step process. First, the node address is resolved into the address of the home node for the object, then that Xanadu node is contacted to resolve the remainder of the address. This two-step resolution is a key contributor to scalability, since it allows each server to control its own address space independent of other nodes, and each server can be under the control of a different organization. It promotes robustness as well, since the outage of a single node does not imply other node contents are inaccessible.

Uniform Resource Locators (URLs) [22] share many of the same properties as Xanadu tumblers, though with an important scalability tweak. URLs take the form:

{scheme}://{domain}{user}/{path}

Here the scheme identifies a specific form of URL, usually associated with a specific network protocol, the domain is the name or IP address of a specific server, the user is user authentication information, and the path is a server-specific identifier of a network resource. Like tumblers, many organizations can each delegate and assign portions of the URL namespace under servers they control. Resolution of a URL is a two-step process. In the first step, the domain name is resolved into an Internet Protocol (IP) address using the Domain Naming System (DNS) [133]. IP address in hand, the client then uses the scheme to determine which network protocol to employ when connecting to the server. If present (and it almost never is), the user information is employed to authenticate the client to the server. Then, the client passes the path portion of the URL to the server for resolution into a resource. Like Xanadu tumblers, URLs are scalable, since each server, and hence organization, controls its namespace independent of other organizations, and robust, since the resources are distributed across a large number of servers. Unlike tumblers, URLs can name objects accessible via multiple network protocols, and URLs employ DNS host names, providing an extra level of naming indirection, permitting the IP address of a server machine to change without altering its objects' URLs.

The Document Management Alliance 1.0 employs addresses called object instance identifiers (OIIDs). OIIDs are actually a form of URL, though their resolution semantics are somewhat different from http and ftp scheme URLs. An OIID has the form:

dma://[{dma pop}]/{system id}/{docspace id}/{object id}[;guid={object guid}]

The beginning string "dma" identifies this as a dma scheme URL, and the dma pop field identifies a DMA point of presence, where a client could connect to a DMA server over the Internet. However, since no network protocol for accessing DMA repositories currently exists, since all access occurs via the DMA Common Object Model (COM) interfaces, and the dma pop field is always ignored. Thus, the standard DNS lookup step in URL resolution does not apply to OIIDs.

The system id field is a globally unique identifier (GUID) that identifies the system that originally assigned the OIID to an object. The docspace id is another globally unique identifier, this time representing a cluster of interconnected systems that are all capable of resolving the OIID. Thus, OIID resolution can be performed by submitting the object id and guid fields to either the docspace, or to the system that originally assigned the OIID. Thus, OIIDs also support scalability, since OIIDs can be assigned by any DMA system, independent of all other DMA systems, and robustness, since a single system, or docspace becoming unavailable does not imply that objects on other systems are unavailable.

Chapter 6

Modeling System Data Models

An essential aspect of a domain model is a description of the relationships between entities in the domain. In the hypertext versioning domain, the fundamental domain entities are works, anchors, links, and various container objects. Relationships between the entities, and the details of which entities belong to which types of container, all vary from system to system, and very few generalizations can be made across the domain. Yet, the specifics of these relationships have a significant effect on the way hypertext versioning systems model typical versioning use scenarios, and on the design spaces for meeting domain requirements. Thus, developing a model of the entities and their relationships is a necessary step both for understanding the behavior of a specific system in the domain, and for understanding new systems developed within the domain.

Containment is by far the most common relationship occurring in hypertext versioning systems. Containers are used to model such abstractions as links, versioned objects, workspaces, and user-defined collections. Thus, determining which entities are containers, and modeling the containment relationships between these containers and other entities, is a primary activity in representing the data model of a hypertext versioning system. Storage relationships, denoting which architectural elements provide storage for elements in the data model, are also important, since they convey information about the control choices in the system. For example, storage relationships in the Chimera system [7] show that Chimera provides storage for anchors, links, and views, but does not provide storage for, and hence does not control, the linked objects.

In summary, the purpose of modeling system data models is:

- Concisely describe the data model of an instance of a system in the hypertext versioning domain using containment, inheritance, and storage relationships between architectural elements and data model elements.
- Allow the data model to expose control choices in the system, making them explicit. This is usually represented by storage relationships.
- Build understanding of the system, so it can then be modified to add hypertext versioning capability. If you don't know what you have, how can you reliably change it?
- Support analysis of system properties, since many system qualities depend on containment relationship patterns.
- Support comparison of systems that have been modeled, allowing similarities and differences to more readily be examined. This also permits new, or contemplated, systems to be compared with existing ones.

The focus of the system data models is on *static*, not behavioral, aspects of the entities and their relationships. In part this is due to the focus of this domain model on versioning static objects and links. However, the primary reason is to abstract away the behavioral aspects to focus on the containment and storage relationships that have such a significant effect on the versioning behavior of the systems. Introducing methods and their parameters into the model acts, in this case, only to obscure the key containment and storage relationships. As a result, an extended entity-relationship model [29], an important member of the class of semantic data models [148,103], is the modeling method used herein for representing the data models of hypertext versioning systems. Entity-relationship modeling was chosen over alternatives such as object-oriented modeling [23] due to its emphasis on static relationships, and the fact that it does not involve modeling behavioral aspects of systems entities, such as methods and their parameters. Semantic data modeling using an enhanced entity-relationship model was successfully employed in the development of the HB1/SP1 system [168].

The essential elements of entity-relationship data models are entities, and relationships [148,103]. Entities signify domain abstractions, such as works, anchors, links, and container types. Typed relationships exist between the entities, and this type is either predefined, such as the "is-a" (inheritance) relationship, or is defined by a specific model. The containment and storage relationships used in hypertext versioning domain models are examples of these. Graphical representations of entity-relationship data models can be made using the intuitive notion that entities correspond to nodes in a graph, while the relationships correspond to arcs. This is the same intuition that underlies viewing a hypertext as a graph, with works as nodes, and links as relationships.

Typically, domain models provide a normative description of domain entities and their relationships. For example, [188] provides a domain model of the theater-ticketing domain that includes relationships such as a theater contains sections, which contain rows, which contain seats, and a performance is composed of a date, time, name, and location. The hypertext versioning domain is much more general than this, because the basic abstractions, work, anchor, and link, can represent a wide variety of concrete representations of objects and their relationships. As a result, hypertext versioning systems tend to have a small number of entities in them, since the basic abstractions have such broad modeling capacity. However, although there are a limited number of relationship types between works, anchors, links, and various container types, the actual permutations of these relationships and entities are many, and not amenable to generalization. As a result, no normative description of domain entities and their relationships is possible in the hypertext versioning domain.

Two factors may contribute to this. First is the preponderance of research systems characterizing the domain. Since these systems were explicitly designed to explore different aspects of the hypertext versioning, it should not come as a surprise that there is no uniform model encompassing all these systems. Second the definition of the domain may itself be too broad. By narrowing the realm of domain modeling, perhaps to one of the categories listed in the domain taxonomy given in Chapter 3, it might be possible to further constrain the allowable relationship patterns. However, even in the narrow category of composite-based systems, there is variation in the kind of containment relationship used, and whether recursive containment is permitted for composites. As a result, the variability appears to be inherent to the domain itself, whether considered narrowly or broadly.

6.1 Modeling Primitives

6.1.1 Entities

A typical entity-relationship (E-R) model uses entities to model data items. In the traditional use of entity-relationship modeling for databases, entities contain a series of attributes, and these attributes are basic data types, such as integers, floating points numbers, and strings. For example, an address entity would be represented as containing street, city, and zip code string attributes. When modeling hypertext versioning systems, entities represent abstractions such as works, anchors, and links. While the concrete representation of anchors and links is similar in granularity to typical entities used in database modeling, the concrete representation of works is much larger, and can be organized according to one of many different internal formats, such as a word processing, spreadsheet, bitmap image, etc. format. As noted in Sections 2.1 and 8.1, objects, which represent works, anchors, and links, typically take one of three organizations, all data, data plus properties, and all properties. The data plus properties and all properties organizations are examples of data aggregation, where the object is composed of one or more properties, and, in the case of the data plus properties organization, a data item representing the contents. Departing from typical E-R diagramming convention, this aggregation of data items is not modeled by having the properties and contents be modeled as attributes. Instead, properties and contents are modeled as entities, and an inclusion containment relationship binds them to their parent entity.

Entities are also used to model high-level architectural elements, such as a server, or a file system. These high-level architectural elements are used when modeling storage relationships, and this use of entities to represent architectural elements is a departure from the typical database modeling use of E-R diagrams. Placing architectural elements and domain abstractions in the same diagram combines together two concerns that are usually separated. Architecture diagrams typically only contain architectural elements, and do not address data modeling issues, and similarly data models typically only contain data items, and do not address architectural issues. By combining them for hypertext versioning systems, storage control choices are highlighted. Making these control choices more visible is an advantage for system architects and designers, since these control choices can have a significant effect on the design of a hypertext versioning system.

Entity-relationship models have entities composed of attributes. This has two drawbacks. One is that it privileges the entity, at the expense of the attribute, rather than treating abstractions uniformly. Second, it creates a special category for the aggregation relationship between entities and attributes, rather than treating it as just one point in a larger design space of containment. Hence, when creating data models for hypertext versioning systems, data aggregation will be represented using inclusion containment relationships with the characteristics of single containment, single membership, no ordering, and with deletion semantics that cause a delete of the container to remove all contents as well (see Chapter 4 for definition of these characteristics).

In the graphical representations of data models, a rectangle will be used to represent an entity.

6.1.2 Relationships

There are three relationship types used when creating data models of hypertext versioning systems, containment, inheritance, and storage. In graphical representations, an arrow-tipped line represents a relationship. Relationships are directional, and exist in both directions. So, for example, a container entity "contains" other entities, which are "contained by" the container.

The containment relationship is used to represent sets of entities. When used in the data model of a specific system, it must be parameterized, to fully define the semantics of the containment relationship. Following the definitions in Chapter 4, the parameters, and their allowable values are:

- **Containment:** single or multiple
- **Membership:** single or multiple
- **Ordering:** ordered, or unordered
- **Containment type:** inclusion, containment relationship on container, containment relationship on object, containment relationship on both object and container, independent relationship object.
- **Container deletion semantics:** deletes container and contents, deletes just the container without affecting contents.
- **Use of a revision selection rule:** whether this containment relationship selects an object from the membership of a versioned object, using a revision selection rule. This containment type is depicted graphically by having the containment arc pass through the versioned object.

Containment relationships have cardinality, depicted as numbers or the letters M and N (depicting more than one, or many) on the relationship, expressing the number of entity *instances* that can exist at each end of the relationship. Note that the number at the container end of the containment relationship must agree with whether it is single containment or multiple containment. Since single containment indicates the entity can only be contained by a single collection, it must be represented by a "1", while multiple containment is represented by M or N, reflecting that the object can belong to multiple containers.

The inheritance, or "is-a", relationship is used to avoid duplication of similar entities in the data model. Entities inherit all of the relationships of their parent, thus avoiding the need to duplicate all of these relationships on each child. Following the graphical convention given in [103], inheritance relationships are graphically represented using a thick double line.

The storage relationship represents that a specific architectural element provides physical storage for an entity. Storage relationships are only used when the storage of entities is split among multiple architectural elements. When only a single architectural element stores all entities in the system's data model, storage relationships are omitted for clarity.

6.2 Data Modeling Examples

As an example of using the this data modeling technique, the data models of the Dexter hypertext reference model [94], Intermedia [205], and NoteCards [190] are given below in Figure 8-Figure 10.

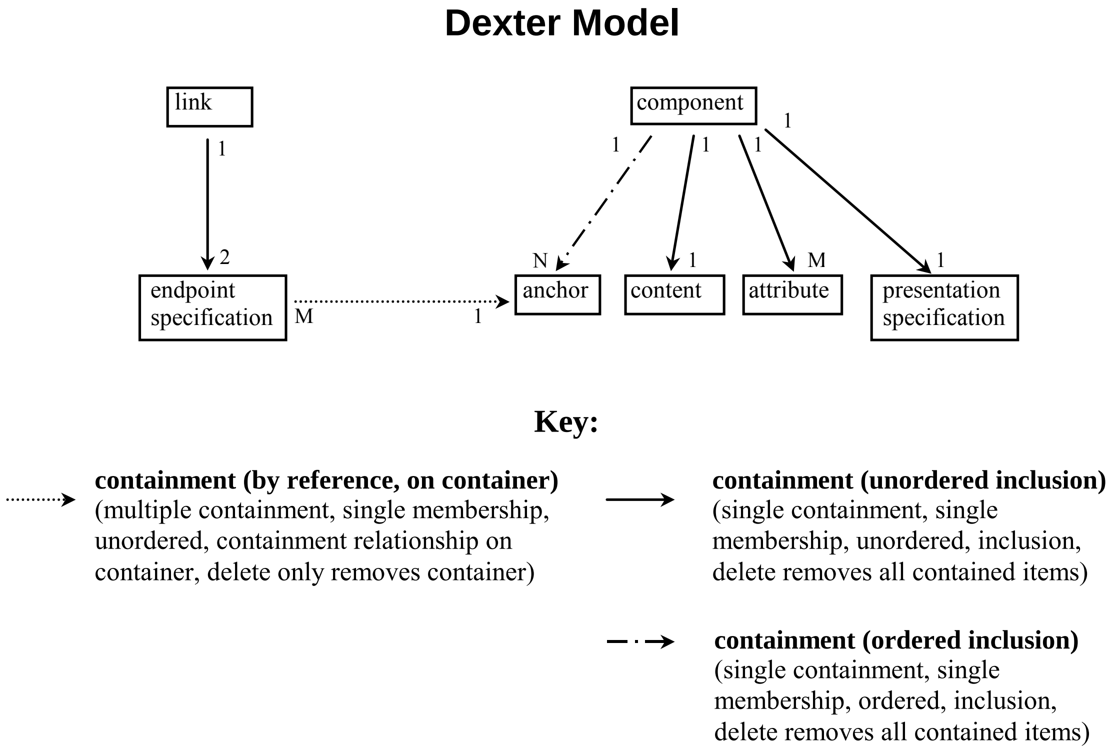
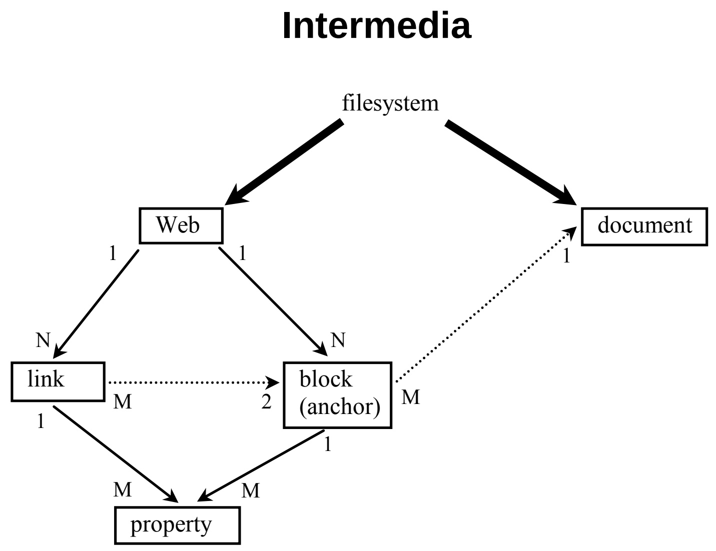

Dexter Model
link
1
2
endpoint specification
M
1
component
1
1
1
1
N
1
M
1
anchor
content
attribute
presentation specification
Key:
containment (by reference, on container)
(multiple containment, single membership, unordered, containment relationship on container, delete only removes container)
containment (unordered inclusion)
(single containment, single membership, unordered, inclusion, delete removes all contained items)
containment (ordered inclusion)
(single containment, single membership, ordered, inclusion, delete removes all contained items)

Figure 8 – Data model of the Dexter hypertext reference model [94].



Key:

⋯→ **containment (by reference, on container)** (multiple containment, single membership, unordered, containment relationship on container, delete only removes container)

→ **containment (unordered inclusion)** (single containment, single membership, unordered, inclusion, delete removes all contained items)

➡ **stores**

Figure 9 – Data model of the Intermedia system [205].

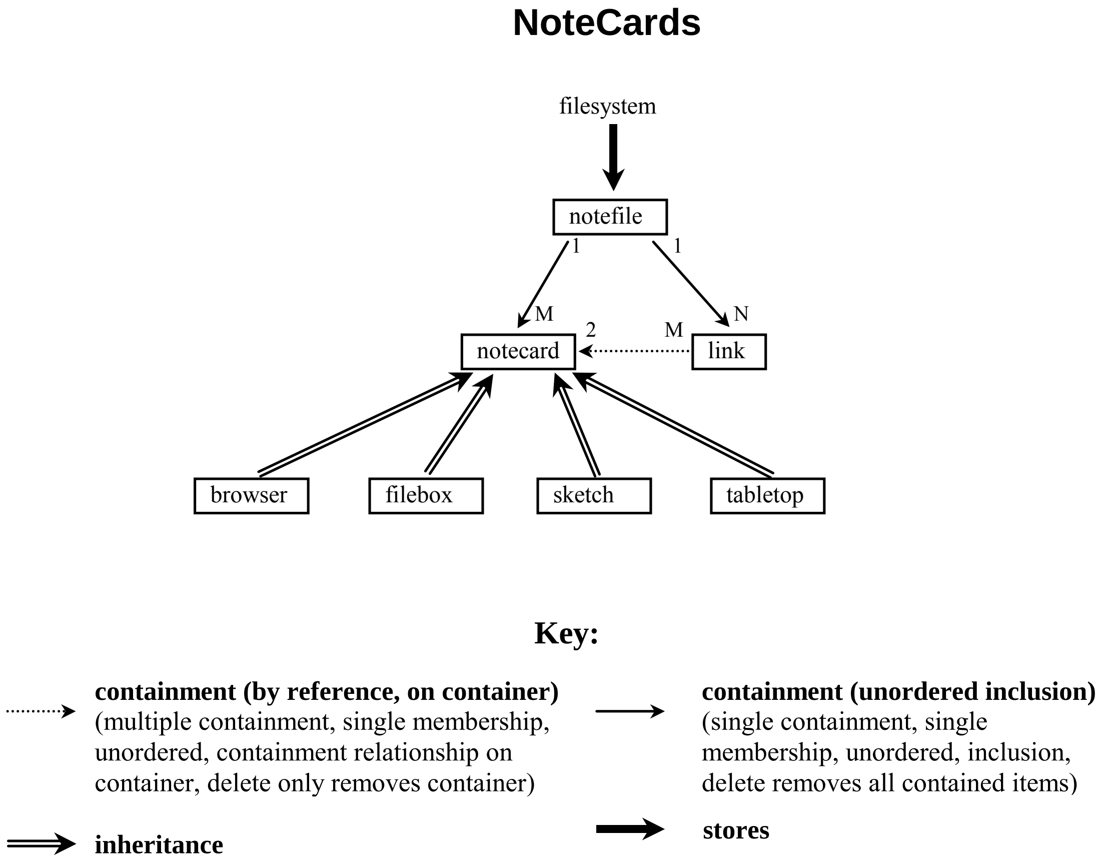

NoteCards
filesystem
notefile
1
1
M
N
2
M
notecard
link
browser
filebox
sketch
tabletop
Key:
containment (by reference, on container)
(multiple containment, single membership, unordered, containment relationship on container, delete only removes container)
containment (unordered inclusion)
(single containment, single membership, unordered, inclusion, delete removes all contained items)
inheritance
stores

Figure 10 – Data model of the NoteCards system [190].

6.3 Relation of Versioning Scenarios to Data Model

The following sections present a common collaborative work scenario that is modeled using the concepts and abstractions of Neptune [46], HyperPro [141], CoVer [87,88], and HyperProp [179]. By integrating multiple data model aspects to represent two authors working in parallel on a small hypertext web, the scenario permits an examination of how data model differences end up affecting the type, number, and composition of system entities over the course of the scenario. The relationships that most affect the behavior of systems throughout the scenario are those of containment. To highlight the interactions between a system's containment relationships and its representation of the scenario, a data model for each system is provided, showing its containment relationships. This is immediately followed by the system's representation of the common scenario.

Figure 11 provides an overview of the scenario, a small hypertext consisting of four documents, connected by five links. Two different authors, "author 1" and "author 2" simultaneously work on the hypertext for a period of time, and then combine their work once they're finished. The scenario involves cases where the same document (D) was modified by both authors, where one author deletes a link (γ) and the other doesn't, where one author creates a link (η) the other doesn't, and where both authors create the same relationship, but as separate links (ζ and θ). For each system, the scenario demonstrates parallel work, changes to hypertext objects, and changes to the hypertext link structure. Where supported, the scenario also shows merging of the authors' parallel work sessions.

Neptune's data model is shown in Figure 12. The most significant difference between Neptune and other composite-based systems is its use of referential containment with single containment, and container deletion semantics that cause a collection, along with all of its members, to be deleted. That is, Neptune provides a type of by-value containment. Due to the use of single containment, objects and links can only belong to a single context, and hence when a new context is created, or a new revision is made for an existing context, all objects and links within the context are duplicated. Thus, it should come as no surprise that Neptune uses more objects than any other system to represent the scenario. Though Neptune's type of by-value containment has the disadvantage of significant duplication of objects and links, it has the advantage of eliminating the need for revision selection rules on links and containment relationships. Since every object and link is contained within a specific context, there is no need to select a specific revision from the pool of objects within a versioned object.

While Neptune's object duplication appears on the surface to make per-item revision history recording difficult, in fact Neptune can record complex version histories. Neptune records a linear version history for each object, and each line of development in the scenario has its own linear history. For example, object A has a main context history, an author 1 history, and an author 2 history. These separate histories are reflected in the version numbering system in the scenario – in the main context, revisions are labeled "M,1" and "M,2", reflecting that these are revisions in the main context history, while in the author 1 context, revisions are labeled "A1,1", "A1,2" to show that these are revisions in the author 1 context. Neptune also records the derivation relationship, so it is possible to follow a revision history from one linear history to another, for example, from the author 1 history back to the main context history (this capability is shown in Figure 7 of [46]).

A quick comparison of HyperPro's handling of the scenario in Figure 16 with that of CoVer in Figure 18 and HyperProp/NCM in Figure 20 immediately highlights a difference in containment. HyperPro contains the entire versioned object (and hence all of its revisions) within contexts, unlike CoVer and HyperProp/NCM who employ containment augmented with revision selection rules to select a single revision from within a versioned object. But, since HyperPro contains its objects by reference, this containment strategy does not lead to a proliferation of objects. The same versioned items, and their revisions, simply belong to multiple containers. In HyperPro, one endpoint of its GenericVersionLink is a versioned object, thus raising the question of which revision is actually displayed after a link traversal. Without providing additional information, there is no rationale for choosing one revision over another. This dilemma is solved by the revision selection rule associated with each context. The revision selection rule causes the GenericVersionLink endpoint ending at the versioned object to select only one of the revisions.

By anchoring a link on a specific revision, and using the context's revision selection rule to pick the endpoint, HyperPro is able to avoid versioning links. As links are object-to-object, it is not a problem if there is a new revision of the endpoint object, since the revision selection rule can be modified to pick a new revision *without changing the link*. Similarly, since the start of a link is associated with a specific object revision, and since links have no anchor points within an object, once this starting revision has been selected, the link is impervious to further object modifications. By implication, if HyperPro links were associated with specific anchor endpoints within an object, this scheme would not work, as the link endpoint location would change from revision to revision at the target object. Unfortunately, when there is a new object revision, all links starting at that object must be duplicated, and, unlike Neptune, there is no revision tracking across these duplicated links. Thus, while the numbering convention in Figure 16 indicates that α, αα, and ααα (shown in the topmost context at time t5) are all revisions of the same link, HyperPro does not record this fact, and cannot display a revision history of this link. They are three separate links to HyperPro. However, despite not versioning links, HyperPro can version structure, since links are contained within contexts, and contexts themselves are versioned.

HyperPro's use of containers shows another drawback. In HyperPro, the first collaborator works on the main branch of the project context, while new collaborators derive revision contexts from the main branch. However, when there is a separate work container for each author, a feature supported by Neptune, CoVer, and NCM, it provides a better separation of work areas than is the case with HyperPro. When there is a separate composite for each collaborator, it is clear where each author's work takes place. Furthermore, by having a separate branch for the state of the system prior to collaboration, it is clear which state of the system new collaborators should use as their starting point.

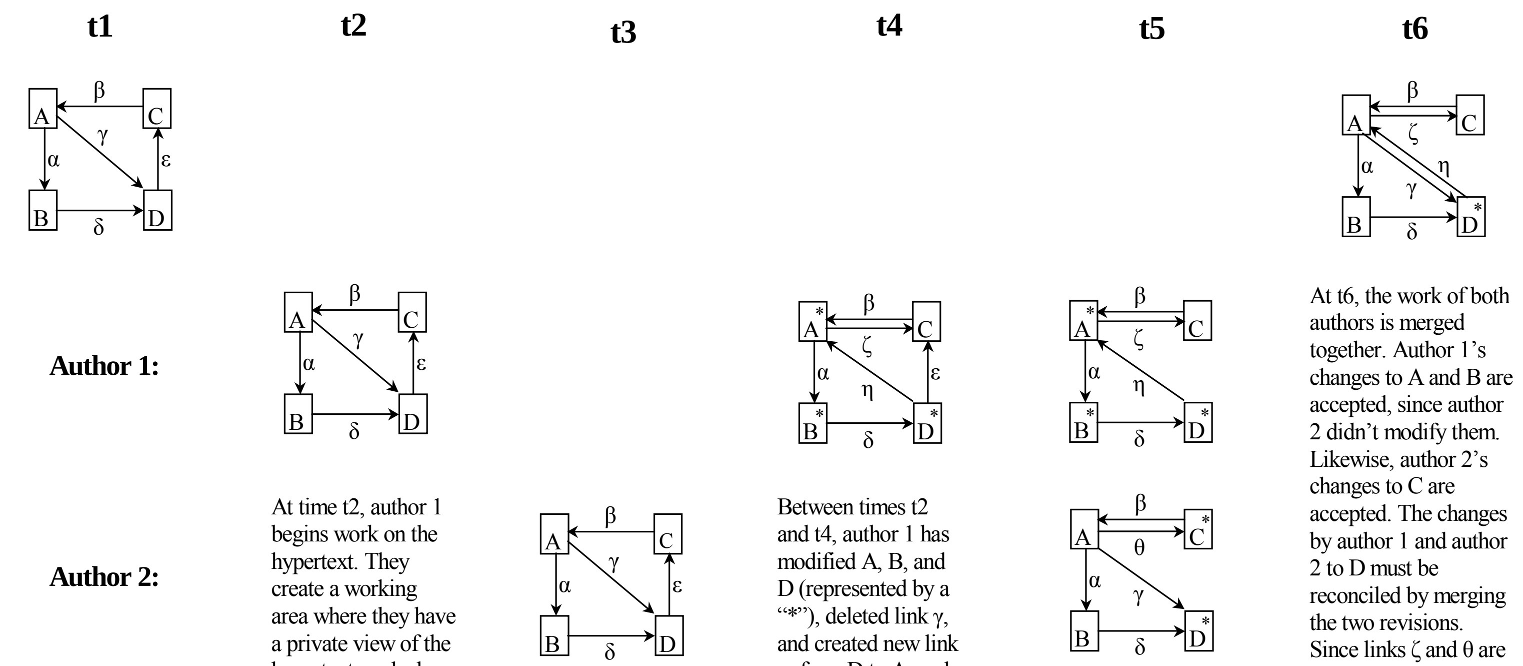
HyperPro's object inheritance diagram in Figure 15 is provided as a contrast to the containment diagram in Figure 14. The object inheritance diagram does provide useful information about the HyperPro data model, such as which abstractions are considered to be specializations of other types. This is useful for noting which types are container types and subtypes. However, the object inheritance diagram has very limited explanatory power for describing how the system models the collaboration scenario. This is directly due to the absence of containment relationship information. In contrast, the HyperPro containment diagram does provide most of the information needed to model the scenario. It shows containers and their contained items, the type and cardinality of the containment relationship, and which items are versioned. It also highlights that every entity has as its state a set of attributes, a fact not evident in the inheritance hierarchy. However, both diagram types still do not capture the entire behavior of HyperPro's data model. For example, even though links are shown to be contained by the HyperPro VersionGroup, this does not imply they are versioned, even though the VersionGroup's containment of objects does imply that objects are versioned. In this case, the links are used within the VersionGroup to represent predecessor/successor relationships, highlighting the need to augment both diagram types with addition explanation of actual use, and constraints.

In summary, the containment diagram has the following benefits:

- Exposes which objects are, and are not versioned, through the presence of versioned object containers.
- Exposes characteristics of links such as their arity (one to one, many to one), and whether they make use of revision selection rules.
- When versioned objects contain links, the diagrams show either that the links are versioned, or the links are used to represent predecessor/successor relationships.
- Increases the visibility of containment cardinality.
- Highlights the container-like nature of links, by modeling links like other containers.

After viewing the complete scenario, CoVer's task view emerges as a valuable simplified view of the ongoing work, especially since CoVer provides a specialized user interface focused only on the task view. By acting as a container that holds all other versioned object types (i.e., the various "mobs"), the task also acts to simplify the containment relationships by not overloading the containment of the organizational container type, the composite. This allows a separation to be maintained between the organizational containment provided by composites, and the workspace containment provided by tasks.

Scenario Overview



At time t1, there are four documents, A, B, C, & D, and five links, α, β, γ, δ, & ε. This baseline state of the hypertext is preserved so it is possible to revert back to this state in the future.

At time t2, author 1 begins work on the hypertext. They create a working area where they have a private view of the hypertext, and where their changes are isolated from other authors. At t2, no changes have yet been made, only the act of creating a separate work area has occurred.

At time t3, author 2 begins work on the hypertext. Like author 1, they create a separate, isolated working area. The initial state of the hypertext is the baseline, from t1. Meanwhile, between t2 and t3, author 1 modifies their view of the hypertext.

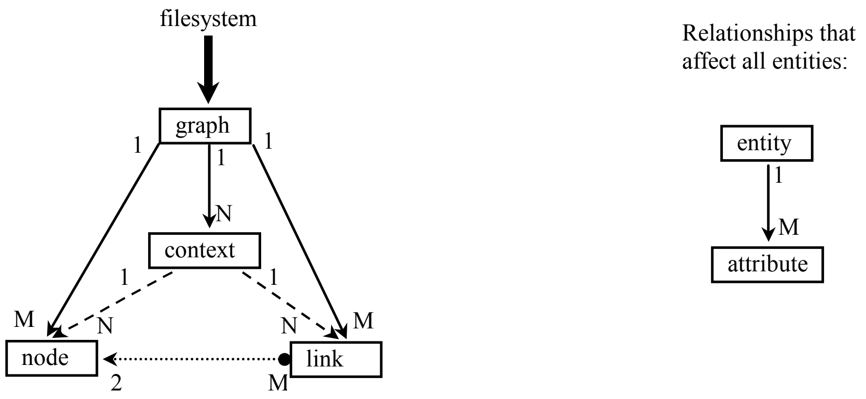
Between times t2 and t4, author 1 has modified A, B, and D (represented by a "*"), deleted link γ, and created new link η, from D to A, and link ζ from A to C. At t4, author 1 acts to preserve the state of document A (but not necessarily of the entire hypertext), so it will be possible to revert to its state as of t4 in the future. Meanwhile, between t3 and t4, author 2 modifies their view of the hypertext.

At t5, both authors stop working. Between t4 and t5, author 1 made further revisions to A, did not modify B or D, created link ζ, and deleted link ε. Between t3 and t5, author 2 modified C & D, created link θ and also deleted link ε.

At t6, the work of both authors is merged together. Author 1's changes to A and B are accepted, since author 2 didn't modify them. Likewise, author 2's changes to C are accepted. The changes by author 1 and author 2 to D must be reconciled by merging the two revisions. Since links ζ and θ are duplicates, they decide to accept link ζ and reject link θ. Author 1 deleted link γ but author 2 did not, so they decide to keep link γ. Author 1 added link η, but author 2 did not, so they decide to keep link η. Both authors agreed to delete link ε.

Figure 11 – Overview of the collaborative work scenario.

Neptune/HAM



Key:

- - → **containment (by value using references)** (single containment, single membership, unordered, containment relationship on container, delete removes all contained items)

●···→ **containment (by reference, on container, ordered)** (multiple containment, single membership, ordered, containment relationship on container, delete only removes container)

→ **containment (unordered inclusion)** (single containment, single membership, unordered, inclusion, delete removes all contained items)

➡ **stores**

Figure 12 – Data model of Neptune/Hypertext Abstract Machine (HAM) [45,46]

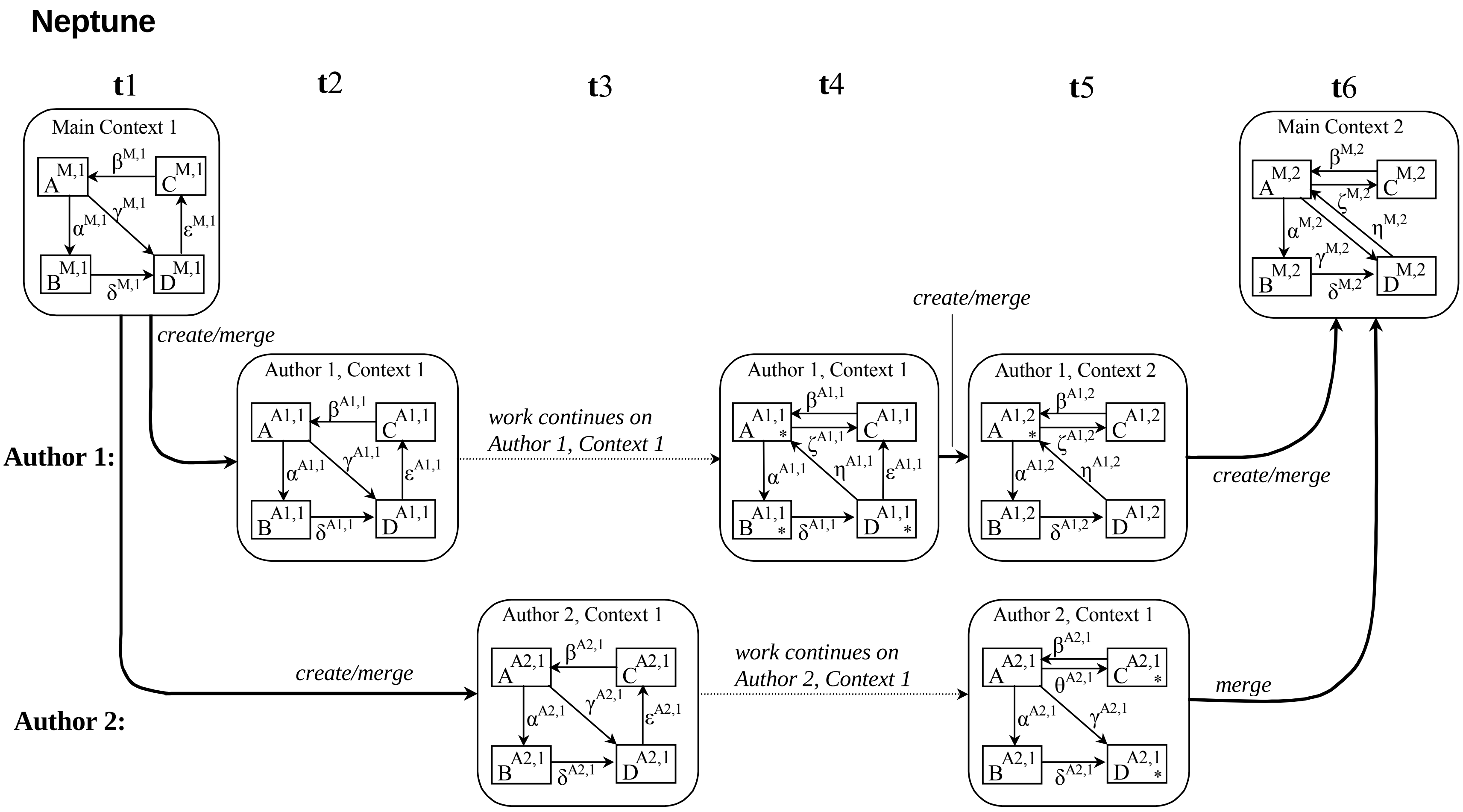


Figure 13 – Common collaborative work scenario, as represented by Neptune/HAM.

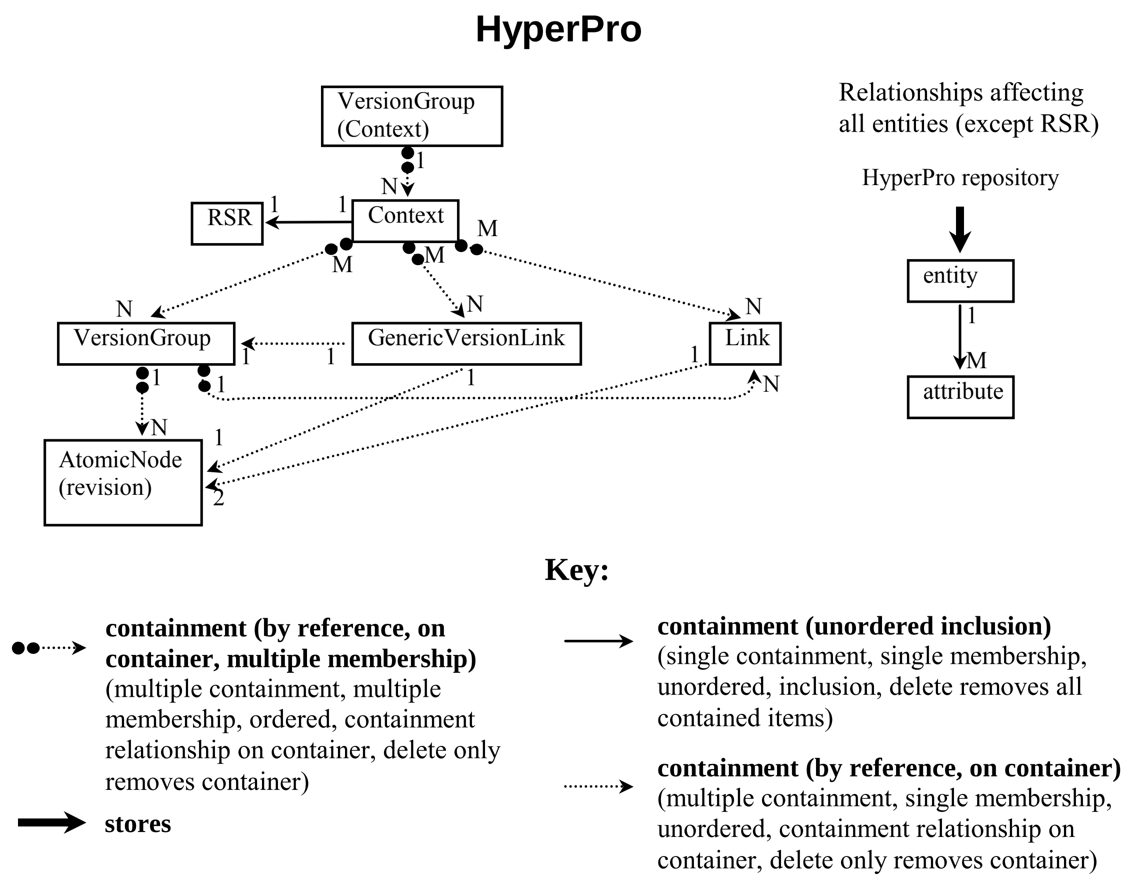

HyperPro
VersionGroup (Context)
1
N
RSR
1
1
Context
M
M
M
N
N
N
VersionGroup
GenericVersionLink
Link
1
1
1
1
1
1
N
N
1
AtomicNode (revision)
2
Relationships affecting all entities (except RSR)
HyperPro repository
entity
1
M
attribute
Key:
containment (by reference, on container, multiple membership) (multiple containment, multiple membership, ordered, containment relationship on container, delete only removes container)
stores
containment (unordered inclusion) (single containment, single membership, unordered, inclusion, delete removes all contained items)
containment (by reference, on container) (multiple containment, single membership, unordered, containment relationship on container, delete only removes container)

Figure 14 – Data model of the HyperPro system [141]. To improve clarity, the full inheritance hierarchy is not depicted on this figure. The RSR is a revision selection rule that selects the revision endpoint of a GenericVersionLink. The existence of separate VersionContexts for objects and contexts represents the constraint that all revisions in a VersionContext must be of the same type.

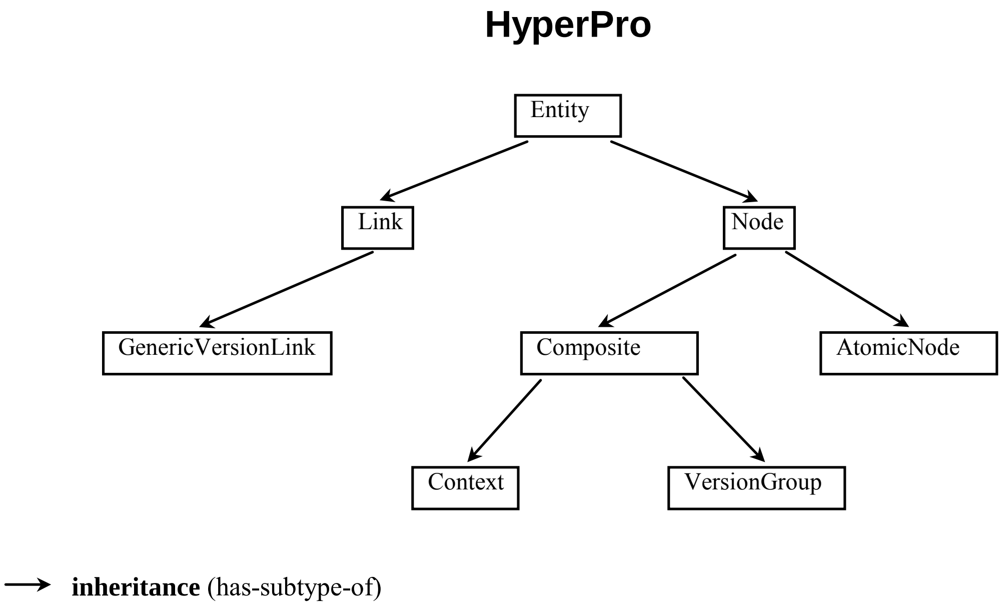


Figure 15 – Object inheritance diagram for the HyperPro system. From Figure 1 of [141].

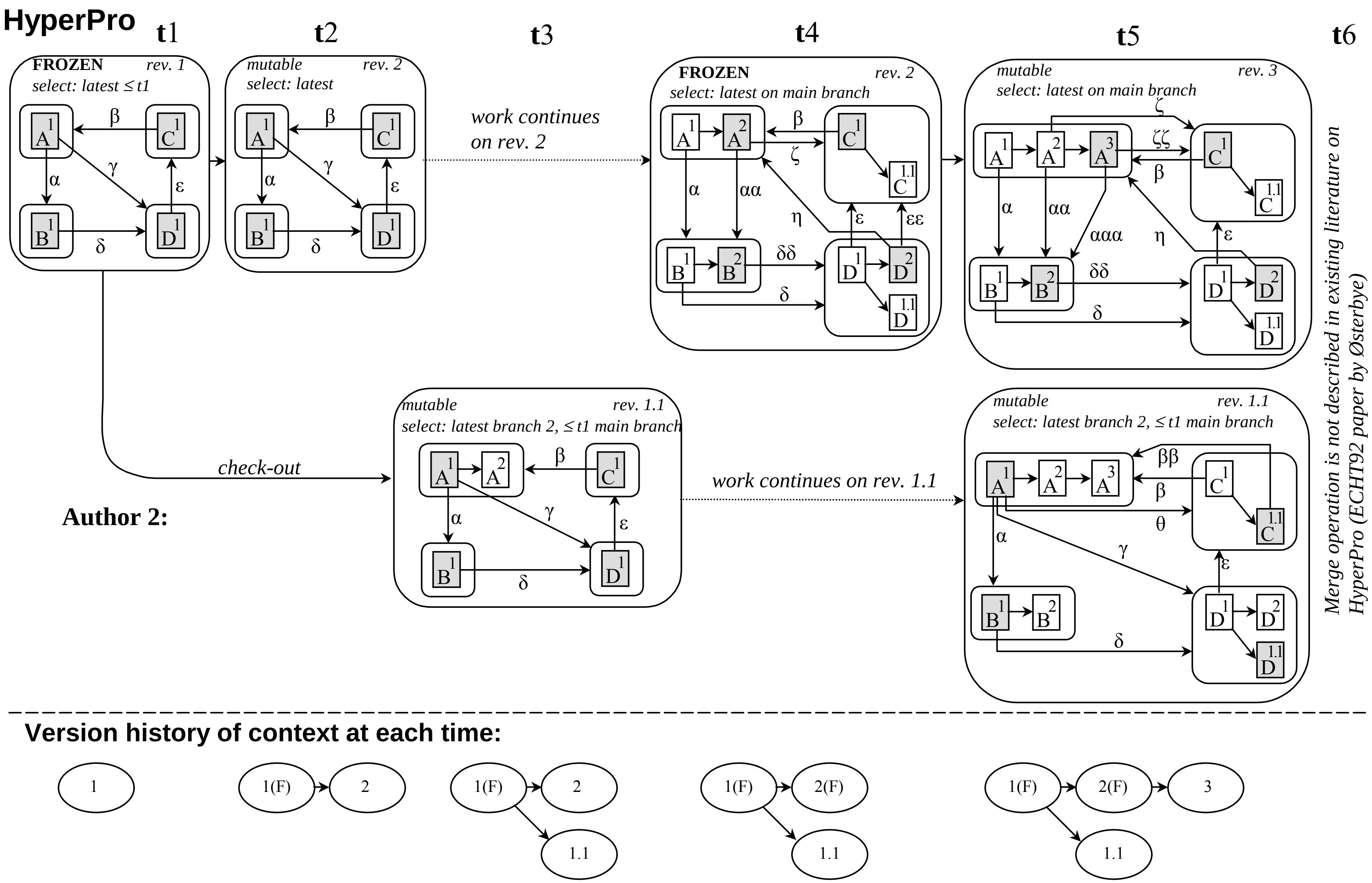
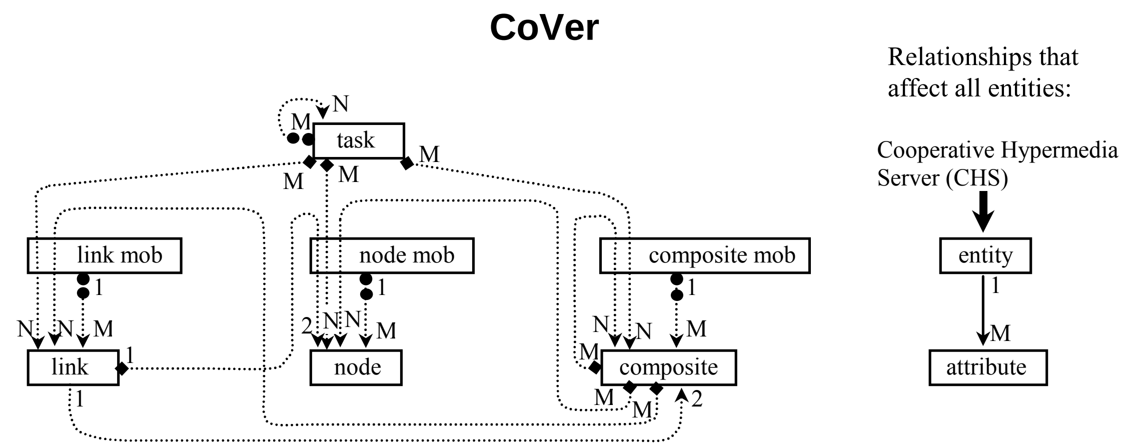

HyperPro
t1
t2
t3
t4
t5
t6
FROZEN rev. 1
select: latest ≤ t1
mutable rev. 2
select: latest
work continues on rev. 2
FROZEN rev. 2
select: latest on main branch
mutable rev. 3
select: latest on main branch
check-out
Author 2:
mutable rev. 1.1
select: latest branch 2, ≤ t1 main branch
work continues on rev. 1.1
mutable rev. 1.1
select: latest branch 2, ≤ t1 main branch
Merge operation is not described in existing literature on HyperPro (ECHT92 paper by Østerbye)
Version history of context at each time:
1
1(F)
2
1(F)
2
1.1
1(F)
2(F)
1.1
1(F)
2(F)
3
1.1

Figure 16 – Common collaborative work scenario, as represented by HyperPro.



Key:

◆·····➤ **containment (by reference, on container, selection using RSR)** (multiple containment, multiple membership, ordered, containment relationship on container, delete only removes container, selected through versioned object using revision selection rule)

⟹ **inheritance**

➡ **stores**

⟶ **containment (unordered inclusion)** (single containment, single membership, unordered, inclusion, delete removes all contained items)

●●·····➤ **containment (by reference, on container, multiple membership)** (multiple containment, multiple membership, ordered, containment relationship on container, delete only removes container)

Figure 17 – Data model of the CoVer system [87,88]. All mobs (multi-state objects) are composite subtypes. While the two referential containment types are listed having multiple membership, this is an extrapolation from [87,88], and these containment types may in fact be single membership. Where containment types involve a revision selection rule to select a revision from the membership of a mob, this is depicted graphically by the revision arc passing through the mob. Since addresses in CoVer consist of an (*object identifier, revision selection rule*) pair, and it is possible to simultaneously have versioned and unversioned objects in the same composite, all containment arcs that involve selection using a revision selection rule can also be containment arcs that *do not* involve revision selection, since it is possible to just omit the revision selection rule from addresses. These additional containment arcs were omitted to avoid visual clutter.

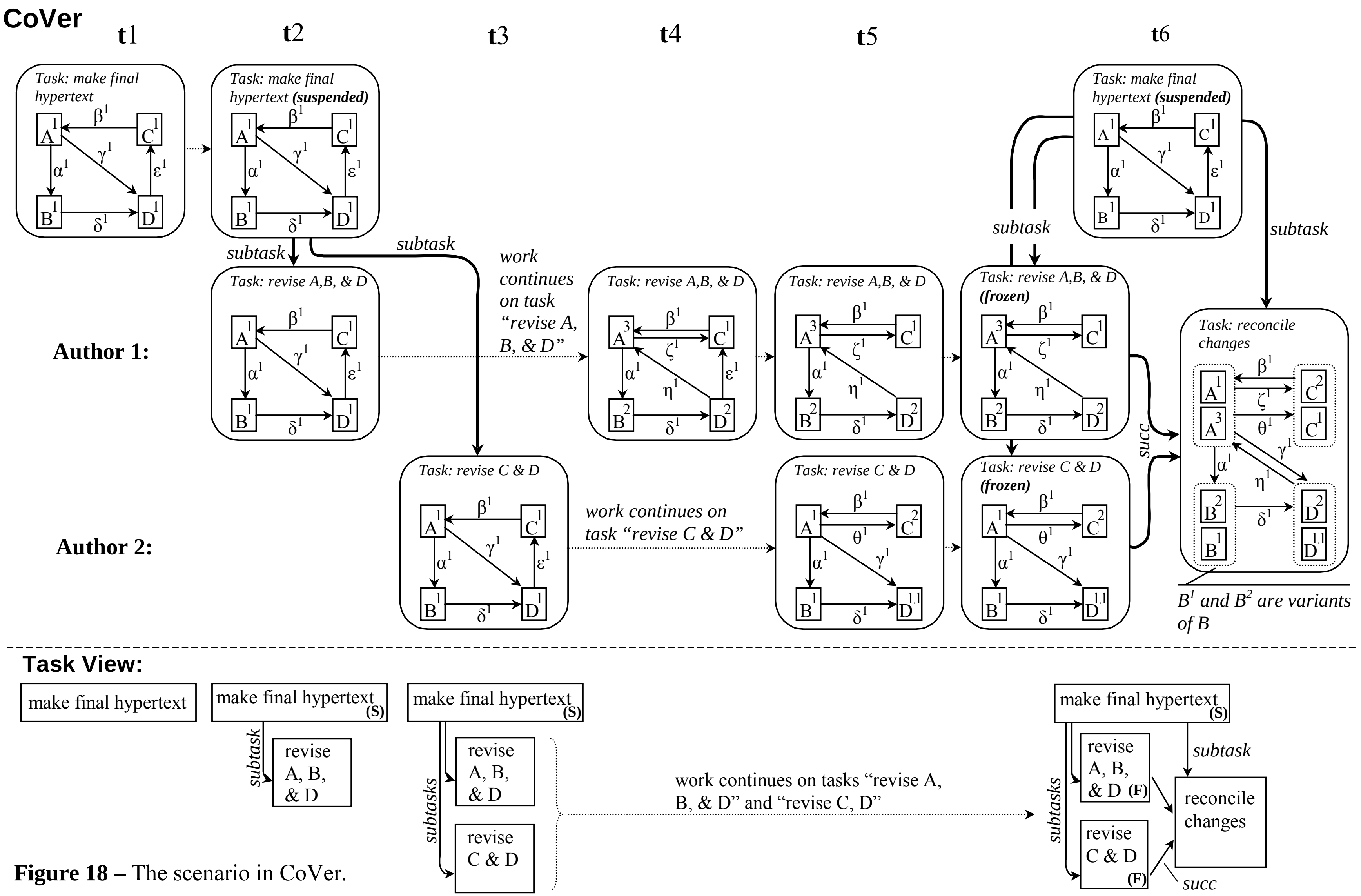

CoVer
t1
t2
t3
t4
t5
t6
Task: make final hypertext
Task: make final hypertext (suspended)
Task: revise A,B, & D
Task: revise A,B, & D (frozen)
Task: revise C & D
Task: revise C & D (frozen)
Task: reconcile changes
subtask
work continues on task "revise A, B, & D"
work continues on task "revise C & D"
succ
Author 1:
Author 2:
B¹ and B² are variants of B
A¹ B¹ C¹ D¹ A³ B² C² D² D¹·¹
α¹ β¹ γ¹ δ¹ ε¹ ζ¹ η¹ θ¹
Task View:
make final hypertext
make final hypertext (S)
subtask
subtasks
revise A, B, & D
revise C & D
work continues on tasks "revise A, B, & D" and "revise C, D"
revise A, B, & D (F)
revise C & D (F)
reconcile changes
succ

Figure 18 – The scenario in CoVer.

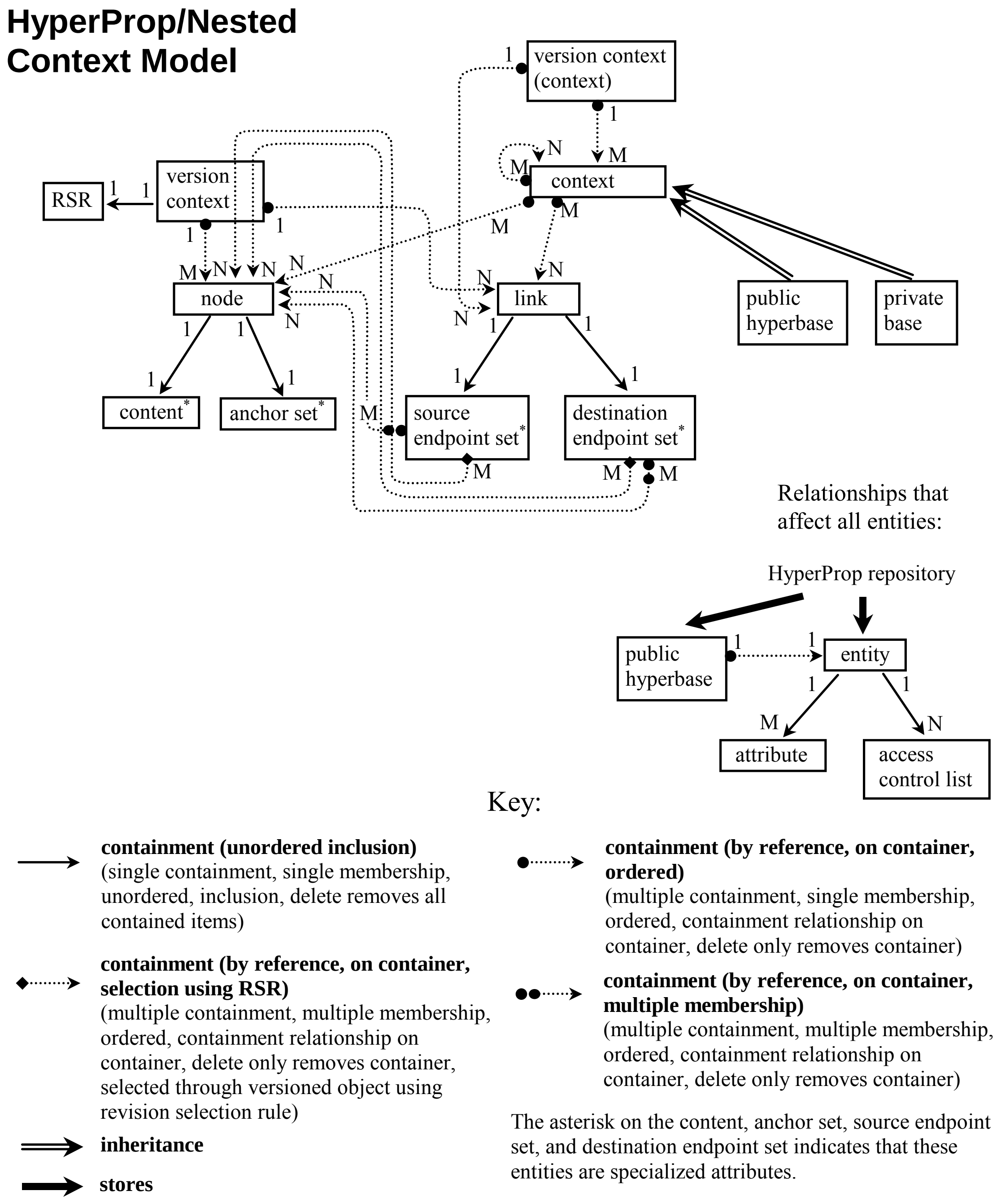
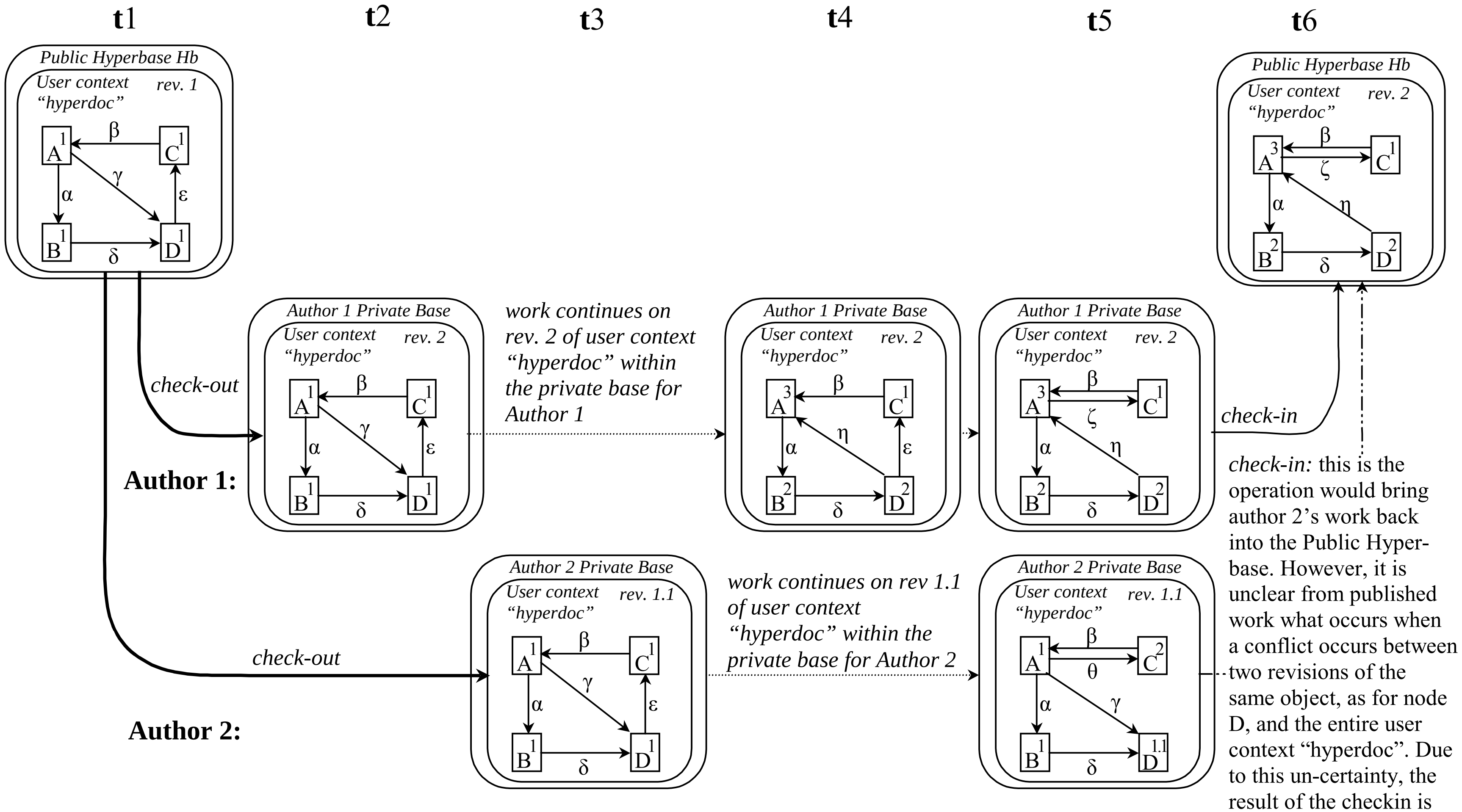

HyperProp/Nested Context Model
version context (context)
context
RSR
version context
node
link
public hyperbase
private base
content*
anchor set*
source endpoint set*
destination endpoint set*
1
M
N
Relationships that affect all entities:
HyperProp repository
public hyperbase
entity
attribute
access control list
Key:
containment (unordered inclusion)
(single containment, single membership, unordered, inclusion, delete removes all contained items)
containment (by reference, on container, selection using RSR)
(multiple containment, multiple membership, ordered, containment relationship on container, delete only removes container, selected through versioned object using revision selection rule)
inheritance
stores
containment (by reference, on container, ordered)
(multiple containment, single membership, ordered, containment relationship on container, delete only removes container)
containment (by reference, on container, multiple membership)
(multiple containment, multiple membership, ordered, containment relationship on container, delete only removes container)
The asterisk on the content, anchor set, source endpoint set, and destination endpoint set indicates that these entities are specialized attributes.

Figure 19 – Data model of the HyperProp/Nested Context Model [179]. The public hyperbase contains one instance of every system entity. The version context is used to represent versioned objects. It seems likely that the version context is not allowed to contain another version context, but this constraint is not explicitly mentioned. Note that contexts are actually a node subtype where the content of the node is the list of contained entities. This implies that contexts also have an anchor set, and the revision selector on the version context resides in one of the anchors of the anchor set. Links contained in version contexts are used to represent predecessor and successor relationships between revisions. This figure contains separate version contexts for representing versioned nodes, and versioned contexts in order to capture the different containment constraints for each versioned object type. However, HyperProp does not maintain this distinction in its data model.

HyperProp / Nested Composite Model



work continues on rev. 2 of user context "hyperdoc" within the private base for Author 1

work continues on rev 1.1 of user context "hyperdoc" within the private base for Author 2

check-in: this is the operation would bring author 2's work back into the Public Hyperbase. However, it is unclear from published work what occurs when a conflict occurs between two revisions of the same object, as for node D, and the entire user context "hyperdoc". Due to this un-certainty, the result of the checkin is not reflected in the Public Hyperbase, and the operation is shown with a dashed line.

Figure 20 – Common collaborative work scenario as represented by HyperProp/NCM.

Chapter 7

Domain Reference Requirements

Systems in the hypertext versioning domain range from the Palimpsest [56] and VTML [194] focus on document format, to systems like DIF [78], HyperWeb [67], Hyperform [202], KMS [4], and Delta-V [199] that version only works, to context-oriented systems like Neptune [46], CoVer [87], HyperPro [141], and the HURL framework [100] that version both works and link structure. These differences in capabilities embody differences in the system's underlying goals. Based on an exhaustive examination of the hypertext versioning literature, this chapter provides an organized list of the explicit and implicit goals of the hypertext versioning domain. Each section below describes a different goal, or related collection of goals, along with any interactions it may have with the other goals. Each section ends with a brief discussion of how its goal interacts with the other goals. Altogether, the goals form the domain reference requirements. Typically, no one system will satisfy all of these requirements. Instead, an individual system will pick and chose from this set of domain reference requirements.

7.1 Data Versioning

7.1.1 The history of objects must be persistently stored.

This is a standard facility of all hypertext versioning systems, and differentiates systems that contain versioning features from those that do not. In essence, this requirement introduces time into the system, adding an extra dimension to objects, composites, and user interfaces. This requirement is explicitly noted in [179,154,41,46,96,125,137], and for the Web in [21,199]. The literature lists many reasons for wanting to record the history of an object, including:

- **Exploration:** allowing for the exploration of the history of evolution of a particular hyperdocument [41] (Section 10.1).
- **Comparison:** comparing two or more revisions to see what has changed between them [137], p. 2/25.
- **Backtracking:** making it possible to see any version of a hyperdocument back to its beginning [46], p. 170. The motivation for backtracking is to maintain previous revisions in case mistakes or wrong decisions need to be undone, to reconsider old choices, or otherwise look at former states [137], p. 2/13. [125] notes that auditability is important for some industries, that is, being able to recover, for legal purposes, prior states of a hyperdocument.
- **Safety:** assuring the safety of recent work against various kinds of accident [137], p. 2/13.
- **Rationale capture:** since the reason for making a particular change soon fades from memory, versioning systems should allow a brief comment to be associated with each change to capture this rationale. Over time, these comments create a group memory for the object [199], p. 197.
- **Reuse:** by preserving a specific revision of a hyperdocument, the entire document, or parts of it, may be reused by others [87], p. 44.
- **Exploratory development:** since authors can depend on the ability to revert to a known, “safe” state of the system, versioning supports exploratory changes, where the final impact is initially unknown [179], §3.1, [141], p. 37.

[179], quoting [96], writes, “Maintenance of Document History – ‘A good versioning mechanism will allow users to maintain and manipulate a history of changes to their network.’” However, it is clear that this combines together the notion of versioning an object and versioning structure. [154] also lists these two requirements together, stating that it is a requirement to “support multiple versions of objects and multiple configurations (i.e., different link arrangements)” (p. 90), though his equating of the term “configuration” with link arrangements differs from current practice in the software configuration management community. However, [101] notes that configurations are the SCM analog to hypertext structure, since a configuration represents the structure of a software system. Perhaps the terms configuration and structure are not so far apart after all.

Interactions with other goals. Several other requirements depend on the ability to record the revision history of works, and other objects. Goals concerning revision mutability (Section 7.1.2), co-existence of versioned and unversioned objects (Section 7.1.3), versioning of all content types (Section 7.1.4), giving human readable names to revisions (Section 7.1.5), change aggregation support (Section 7.3), linking to a specific revision (Section 7.4.3), navigation in, and visualization of, versioned spaces (Sections 7.7 and 7.9), as well as namespace interactions due to versioning (Section 7.14), all are predicated on works being versioned. Recording the revision history of works adds complexity to a system.

7.1.2 Mutability of primary state and metadata for object revisions must be supported.

At its core, this requirement is concerned with just how much the past can be changed. In systems with immutable revisions, all revisions start out as a mutable object, and then become immutable once a checkin operation is performed. Thus, the issue of mutability concerns the ability to modify objects that the user has, at some point, declared to no longer be capable of changing. As opposed to the true past, which can never be changed, the past in versioning systems is stored persistently by the computer on some writeable medium, and can be modified. The literature notes several reasons for wanting to change stored revisions:

- **Linking:** If anchors are stored within an object, and linking to/from frozen objects is allowed, then some mechanism for temporarily making an object mutable is required to support linking. [141], p. 35.
- **Modifying system-specific metadata:** Since system-specific information, such as access control lists, are often stored as metadata, it is useful to be able to modify this metadata after its state has been frozen. However, some metadata specifically depends on the object's value (e.g., an attribute listing the size of the object), and should only be modified if the value is too. It might also be attractive to add new attributes to an object. [141], p. 35, [199], p. 196.
- **Making a small change:** If there is a minor, non-semantic change to a human-readable document, such as a spelling error, it may be reasonable to allow the error to be fixed rather than creating a new revision. [199], p. 193.
- **Preserving logical revision names:** In some cases it is important to maintain the logical name of a revision, especially in cases where some external authority controls revision names. For

> example, the Internet Engineering Task Force assigns revision identifiers to working drafts. If a spelling error is found in one of these drafts, it is preferable to fix the document without creating a new revision, since that would imply a new official revision had been made. [199], p. 193.

The literature is far from unanimous on this capability. Though [141] and [199] agree that some object metadata must remain mutable, while other metadata is frozen with the object contents, they differ on whether the object contents can be immutable. [141] states, “it might be too simplistic to have versions of objects be completely immutable. While it is obvious that the contents of a version (i.e., a frozen object) should be immutable, it is less clear how links and attributes should be treated.” (p. 35). However, [199] has requirements stating both that, “some properties on revisions may be changed without creating a new revision” (p. 196), and that, “revisions may be mutable or immutable.” (p. 192).

Interactions with other goals. Mutable revisions allow the past to be altered, and this disturbs several other capabilities. Mutable revisions make it difficult to reliably version composites, or other forms of structure (Section 7.4.2), since work revisions contained by a composite revision may change, thus altering the meaning of the composite. Similarly, the endpoint of a link might change, thus altering the meaning of the link, thus affecting structure and link versioning (Sections 7.4.1 and 7.4.2), as well as the ability to create stable references (Section 7.2). The ability to change revisions increases the value of tracing the use of parts of compound documents (Section 7.10). If a compound document includes a mutable revision as a sub-part, then it is possible the document could change if the revision is changed.

7.1.3 Versioned and non-versioned objects can coexist

In a hypertext, it is possible that a user may desire to version some, but not all of the objects. In this case, versioned and non-versioned objects coexist in the same hypertext or container [87], p 44. Allowing this increases system complexity, since containers—either composites, or directories in a hierarchically organized namespace—now have to handle the issue of whether a versioned container should record the membership of an unversioned object when its state is frozen. If it does record the membership, it is possible that, in the future, the unversioned objects will be deleted, and the prior container state will now contain dangling membership relationships. If, on the other hand, the unversioned objects are not recorded as part of the container’s state when it is frozen, containers are now complicated with two kinds of membership, versioned and unversioned. Furthermore, when a versioned container is reverted to a prior state, the system is faced with the choice of either removing all unversioned objects from the container, or preserving the unversioned items and only reverting the versioned objects, making it so that the exact prior state of a container can never be recovered. Due to these complexities, hypertext versioning systems that version containers adopt an all or nothing approach, requiring that all of a versioned container's items be versioned.

Interactions with other goals. As noted above, allowing containers to hold versioned and unversioned objects increases the complexity of versioning these container types, affecting structure versioning (Section 7.4.2) and link versioning (Section 7.4.1). Additionally, coexistence of versioned and unversioned objects in collections affects the interactions of versioning with collection-sensitive namespaces, such as filesystem paths (Section 7.14), and would presumably impact some visualizations of the space of versioned and unversioned objects (Section 7.9), and user's interactions with that space (Section 7.11).

7.1.4 All content types must be versionable

Since hypertext systems allow for browsing, editing, and linking between all types of works, including text, images, movies, sound, etc., a hypertext versioning service must be capable of versioning these disparate information types [199], p 191. Furthermore, since many document and image types undergo constant evolution (e.g., the many versions of the Word document type, and HTML), remaining independent of content type ensures the versioning facilities can accommodate this evolution. In contrast, supporting only text versioning, or tailoring versioning facilities too tightly to one, or a small set of content types, increases the brittleness, and reduces the applicability of versioning facilities.

Interactions with other goals. Providing versioning services for a wide range of content types effectively limits the available options for versioning, variant support, concurrency control, and merging (Sections 7.1.1, 7.5, and 7.6). Versioning and variant support requires that the system either has knowledge of all content types used, or delegates version and variant management to each individual application, or uses approaches that do not require knowledge of the object's structure. Typically the latter is chosen. Concurrency control and merging have the same choice: either build-in knowledge of all content types to take advantage of techniques that allow collaborators to simultaneously work on subsections of the object, or choose techniques, such as locking, that require no such knowledge.

7.1.5 A mechanism must exist for giving a human readable name to a single revision.

Frequently, a specific revision of a versioned object has significance beyond just being an intermediate state of an evolving object. A revision may have been released to a customer, or submitted for external review. For these revisions, it is useful to assign a human readable name to that revision, such as "customer release" or "external review" [199], p. 198, [125], p. 21. These human readable names are often called *labels* in configuration management systems, and can be used in revision selection rules.

Interactions with other goals. Once human readable names are available, it increases the value of searching (Section 7.8), since now revision selection rules can be created that select the revision matching a name. The ability to set a human readable name on a revision depends on at least some piece of metadata still being writeable (Section 7.1.2). Human readable names are often displayed as part of visualizations of revision history trees (Section 7.9).

7.1.6 Revisions and versioned objects can be removed.

Over time, revision histories of frequently modified objects can grow large, and older revisions in the history are no longer relevant, or accessed. In this case, it is desirable to be able to delete older revisions, so that their storage can be reused [5]. Alternately, an organization's record keeping policy might dictate that documents are destroyed after a specific period of time. Since it involves destruction of the past, and can lead to the inability to reconstruct prior states of versioned containers that hold the deleted object, this is generally a high privilege operation. As revision identifiers are typically unique across a revision history, the revision identifier of a deleted revision cannot be reused.

Interactions with other goals. Similar to allowing the primary state of a revision to be modified, deletion of revisions and versioned objects modifies the past, with pervasive effect on other features. Deletion negatively affects the ability to reliably reconstruct prior revisions of composites, and other versioned structure containers (Section 7.4.2). Change aggregation, since it involves containment of revisions, is affected if a revision is removed (Section 7.3). If the endpoint of a link is deleted, it leaves the link dangling, and prevents reference stability (Section 7.2), as well as linking to the deleted revision (Section 7.4.3). If revisions are marked as deleted, but their data is not expunged from the system, it may be possible for revisions to be returned in the results of a query (Section 7.8). Use tracing facilities could be used to determine whether there are any usages of a revision slated for deletion (Section 7.10). Deletion has a significant negative impact on the consistency of structures (Section 7.15).

7.2 Stability of References

A major goal for hypertext versioning is to preserve prior object states to ensure that links never dangle, a goal explicitly noted in [137], p. 2/25, [194]. The underlying assumption is the linked object state will always be available, since it is persistently stored, and hence the link endpoints will, likewise, always be present. Both [137] and [194] assume that the owner of the information is responsible for storing, and incurring the costs of maintaining prior states. However, there are many valid reasons for the owner of a document to permanently remove it, and all its prior states. For example, in corporate settings, mergers and acquisitions can make a company name obsolete, product lines can be modified or terminated, in both cases making it desirable for the owner to remove all documents that reference the old company or product names. As a result, reference stability can only be achieved by incorporating third party versioned document stores, where the third party has no compunction about keeping around older information. That is, the implication of a third party caching old company or product names is different from the owner preserving this state. For the third party, no endorsement of the older states is implied, but for the owner of the information, the mere fact of making older information available increases its perceived value.

Interactions with other goals. Ensuring the stability of references requires that the contents of revisions are immutable (Section 7.1.2), so that link endpoints (and other references) retain their meaning. Updating references depends on the ability to retrieve later revisions of a work (Section 7.1.1). If the reference is to a part of a compound document, use tracing of that part is valuable when updating references (Section 7.10).

7.3 Change Aggregation Support

It must be possible to group a set of changes that together constitute a coordinated, logical change, a goal noted in [87], p. 45, [154], p. 91, [179], §3.1, and [199], p. 197. In state-based versioning systems, the set of revisions that together comprise a logical change is called an *activity* [199], a *task* [87], or a *version set* [179]. In change-based versioning systems, sets of changes are, appropriately enough, called *change sets*, while PIE [80] calls them *layers*. There are several reasons for aggregating changes:

- **Labeling a set of changes:** The ability to give a name to a set of changes [199], p. 197.
- **Tracing changes to revisions:** Since a logical change can span multiple versioned objects, long after a change was made it can be difficult to reconstruct which revisions implement a change. By recording which revisions comprise a change, it is possible to trace a change to its revisions, and vice-versa [179], §3.1.
- **Combining version sets:** In change-based systems, changes are recorded so each logical change can be managed as a single entity. Changes can then be combined to create new states of the hyperdocument [154], p. 91.

Interactions with other goals. The ability to label a set of changes might involve the same facility used to record human readable names on individual revisions (Section 7.1.2). It is possible that the use tracing mechanism (Section 7.10) could also be employed for associating a problem description with a logical change. Change aggregation depends on changes being persistently recorded (Section 7.1.1).

7.4 Link and Structure Versioning

It may seem strange to separate versioning of links from versioning of structure. After all, one might expect that structure versioning would be an emergent property of versioning a set of links. Because HyperPro [141] has demonstrated that it is possible to version structure without versioning links by placing the unversioned links inside a versioned composite, and since Hicks [100] goes to great lengths to separate structure versioning from object versioning, we intentionally separate the goal of versioning links from the goal of versioning structure.

The reasons for versioning links and structure are the same as for versioning objects. That is, links and structure are versioned to support revision history exploration, backtracking, safety, rationale capture, reuse, and exploratory development.

7.4.1 It must be possible to version links.

Since links have state, such as their endpoints, or additional metadata, it is possible for the value of a link to change over time, and hence it is desirable to record important points of its evolution. In particular, when a an object representing a work is frozen, it is desirable to permanently record link endpoints so that when it is reverted to this state at some point in the future, the links will be available in their original form [46], p. 170.

Interactions with other goals. In order to independently version links, the links must not be inclusively contained within the works they link, and hence subject to object versioning (Section 7.1.1). The existence of versioned links can increase the complexity of visualizations and user interfaces (Sections 7.9 and 7.11), and interactions of tools with the hypertext system (Section 7.12).

7.4.2 It must be possible to version structure.

Several hypertext systems, including Microcosm [43], Chimera [6] and the Hypermedia Version Control Framework [100], allow multiple sets of links, or structures, to be applied to the same set of works. Due to the ability to apply multiple structures the same set of works, and the separation of versioning responsibility between work and link versioning in link server architectures, it is desirable to version structure independent of versioning works. The versioning proposal presented in [200] shows links versioned separately from data. Meeting this goal is very challenging, as Hicks notes when he states, "The requirement to version structure is one of the main challenges which hypertext brings to the version control area." [101], p. 13.

When works and links are inextricably combined into composites, as is the case in Neptune [46], HyperPro [141], HyperProp/NCM [178], CoVer [87], and VerSE [91], it is still desirable to version structure, although it is no longer possible to separate versioning of structure from versioning of works.

Interactions with other goals. Structure versioning that includes works in the structure, such as versioning of composites, depends on works being versioned (Section 7.1.1), and their contents immutable (Section 7.1.2). When structures contain their objects using revision selection rules, it adds a dependency on searching (Section 7.8), possibly in conjunction with the use of human readable names (Section 7.1.5). Structure versioning ideally accommodates variants, if present (Section 7.5). Structure versioning affects visualizations of the versioned space (Section 7.9), both adding complexity due to the presence of structure revisions, and reducing it due to the modularization of the hypertext (e.g., composites can hold a subset of a hypertext, and hence can be individually visualized). Similarly, tool interactions (Section 7.12) can be more complex due to multiple structure versions, and alleviated through careful defaulting of revision identifiers when a particular structure revision is in use, as was employed in the addition of structure versioning to Neptune/HAM [46]. Structure versioning increases the difficulty of reverting just a single work revision contained within a structure (Section 7.15).

7.4.3 It must be possible to link to a specific revision.

Linking to specific object revisions makes it possible to construct works that always link to a specific revision. For example, if one document is describing the evolution of another, within the text that describes each revision there should be a link to that revision. Linking to a specific revision is a standard facility of hypertext versioning systems, and hence is not typically mentioned, although it is explicitly noted as a goal in [199], p. 196.

Interactions with other goals. The ability to link to specific revision of an object depends, of course, on the objects being versioned (Section 7.1.1). Additionally the user interface must make it possible to create a revision-specific link (Section 7.11). This could be an issue if the user interface restricts link creation to only those revisions visible in a composite revision, since creating a link to an arbitrary revision would require the inclusion of that revision in the composite. Creating a link to a specific revision also depends on each revision having a distinct identifier, which is then used either by an anchor, or a link (Section 7.14).

7.5 Variant Support

There are many types of object variants, including alternate revisions that were developed during parallel collaborative work, translations into different human languages, and renditions of the content as different content types (e.g., PDF and Postscript) [137,179]. Structural variants are also possible, representing alternate structures for the same set of content [179]. A hypertext system should handle these different kinds of variants, by providing a data model and operations that can represent and manipulate variants.

Hypertext versioning systems also support literary use, representing multiple variants of the same work over a period of time. Machan writes, “hypertext enables editors to assemble in one edition all the versions of a given literary work and readers to access these versions, or parts thereof, in any number of ways.” [122], p. 303. Ideally, the facilities used to represent other kinds of variants will have the expressive power to model alternate versions of complex documents, such as the Canterbury Tales discussed by Machan. Hypertext systems should also be capable of allowing authors to create new alternate versions, not just represent existing ones, even when this involves a change in ownership, authorial control, or media [137], p. 2/39. If an object does get developed into an alternate version, or is reused in a different context, it should be possible to trace this use, either from the new context to the original, or vice-versa [87], p. 45.

Interactions with other goals. Frequently, the approach chosen for representing variants affects or extends that used for object versioning (Section 7.1.1). Additionally, it is unusual for a system to provide variant support and not also provide version support. Revision history branches have a dual use, representing variants, and also for concurrency control (Section 7.6), with each collaborator’s changes kept on a separate branch. Traceability features can be used to detect other variants of a work (Section 7.10). Variants affect user interfaces and visualizations as well (Sections 7.11 and 7.9).

7.6 Collaboration Support

Versioning makes it possible for two or more people to work on the same document or link simultaneously, by storing each person’s work in a separate revision. As a result, versioning systems should support collaborative work by providing independent work partitions that allow concurrent authoring of the same information [179], while preventing one author from interfering in the work of other collaborators [46], p. 174-175. The ability to work in isolated areas implies a need to merge together several changes once parallel work has ceased [90,154], p. 91.

[90] notes two additional goals specifically related to merging hypertext networks:

- It should be possible to select a merge procedure, based on the hypertext data model and group work situation.

- Interactive merge tools will be required, since, in general, it will not be possible to automatically resolve merge conflicts. These tools should provide the ability to specify merge results interactively.

In addition to isolating changes, versioning supports collaboration by acting as an awareness mechanism. For example, [89] notes that in the absence of versioning, "… co-authors rejoining work e.g. after holidays found it difficult to find out what happened in the meantime." (p. 408). Comparing revisions provides awareness of a collaborator's modifications. Examining an item's history shows who has worked on it, and hence who can answer questions about it, while stored comments provide awareness of change rationale.

Interactions with other goals. Revision history branches are sometimes used as a concurrency control technique, but can also be used to represent variants (Section 7.5). Composite based hypertext versioning systems use the composite both for versioning structure (Section 7.4.2), and as a workspace, isolating each collaborator's changes. In software configuration management systems, workspaces are often made available via the filesystem, and hence have namespace interactions (Section 7.14). The range of available concurrency control techniques, and the number of content types that must be understood by a merge facility, both depend on whether all content types must be versionable (Section 7.1.4). Concurrency control techniques that allow multiple collaborators to work on object sub-parts simultaneously (i.e., the operational transformation, sub-object replication, or sub-object locking schemes discussed in Section 8.7.1) substantially increase the effort required to integrate existing tools with the hypertext system (Section 7.12).

7.7 Navigation in the Versioned Space

A hypertext versioning system should provide the ability to perform hypertext navigation both within a consistent time slice (i.e., across multiple works at the same time), as well as forward and backward in time [137], p. 2/26. It must be easy for people to access different revisions of a work [180]. Readers should know when they are interacting with a work that is under version control [180]. Since link traversals across time are possible, a hypertext versioning system should provide information to the user so he knows what time each visible object represents, and what time lies at the destination of a link traversal. It should also be possible to select between navigation in a consistent time slice, and navigation always to the present revision. Generalizing, it would be desirable to have the user select the version selection criteria used in link traversals. For example, navigation based on non-time based criteria should also be possible, such as queries on attribute values [87], p. 45.

Interactions with other goals. The capabilities described above depend on the existence of work revisions (Section 7.1.1), and also structure versioning (Section 7.4.2). In an open hypertext architecture, providing the destination revision of a link before it is traversed requires tools to retrieve this information, and then present it in their user interface (Sections 7.12 and 7.11). Navigation using version selection criteria involves the use of searching facilities (Section 7.8).

7.8 Searching

In addition to hypertext link navigation, searching is a useful mechanism for accessing desired objects. CoVer notes this goal, stating, "similar to accessing objects by posing a query, that author wants to access versions of hypertext objects on the basis of their attribute values," [87], p. 45. Query capabilities increase the value of metadata, since it provides a way to select objects based on this metadata, rather than just using the metadata for storage.

Additionally, as noted in Section 4.6 on dynamic containment, queries are also useful as a means of identifying individual containment relation endpoints (CoVer [87] and the Hypermedia Version Control Framework [100] are two examples), and for populating the entire contents of a container (DHM [84] has this capability). Revision selection, where a specific revision is selected from a versioned object, can also be viewed as a restricted form of query.

Interactions with other goals. A search facility often accompanies the ability to set human readable names on revisions (Section 7.1.5). Revision selection rules are frequently employed in structure versioning (Section 7.4.2). Revision selection rules can also be employed when a single location in a namespace needs to map to one revision in a revision history (Section 7.14). Exposing the full capabilities of a query facility through a user interface can increase the perceived complexity of a system (Section 7.11).

7.9 Visualizing the Versioned Space

Nelson captured one facet of this goal when he wrote, “while the user of a customary editing or word processing system may scroll through an individual document, the user of {Xanadu} may scroll in time as well as space, watching the changes in a given passage as the system enacts its successive modifications.” [137], p. 2/18. Unfortunately, this goal of dynamic, animated difference visualization has not been implemented by any hypertext versioning system to date. A more achievable goal is to visually display the differences between two textual revisions, a capability provided by Neptune [46], CoVer [87], HtmlDiff [51] and TopBlend [30] for HTML Web pages, and strongly advocated for by Xanadu [137], p. 2/20. Since hyperdocuments often contain non-textual elements, such as graphics, video, etc., techniques for visualizing differences between multiple revisions should also be provided [90]. Similarly, comparing the differences between two variants or alternate versions should also be possible.

A hypertext versioning system should also provide visualizations of a single object’s revision history, and of an entire versioned hypertext. Durand and Kahn, in their description of the MAPA visualization system for unversioned Web sites, describe several goals that apply to visualizations assisting navigation in versioned hypertexts. These include, “show where I am,” “show where I have been,” “show where I can go from here,” and, “show how I got here” [55]. Many configuration management systems have created graphical visualizations of versioned items, and within the hypertext versioning literature, CoVer [87] and VerSE [91] provide such a visualization. Unfortunately, they reverse convention by having arrows pointing to predecessors, rather than successors, and hence the arrows in their diagrams point opposite the flow of time, making it easy to mistake the youngest revision for the oldest. Furthermore, the visualization of a versioned item should display incoming and outgoing links, along with their revision selection criteria.

The task view in CoVer is a significant innovation, since it provides a visualization of just composite evolution, abstracting away their contents. As Figure 18 (on page 93) highlights, showing both the composite and its contents results in space intensive, busy displays. Providing a visualization of only a composite as it evolves should be a goal of composite based hypertext versioning systems.

Though not intended as a visualization paper, the figures in Perry, Siy, and Votta’s paper on parallel changes in the 5ESS system show several examples of revision history visualizations that would be useful for managers of a software development process [149]. In particular, their graph of deltas per month for each software release (Figure 1), the large-grain visualization of lines changes from revision to revision (Figure 6), and the chart of change requests per file (Figure 12) all seem very useful, and could potentially extend from code files into the realm of documents.

Merging together hypertext networks raises more needs for visualization. When merging hypertext networks, it is necessary to merge the hypertext structure in addition to the more pedestrian problem of merging objects. In order to merge hypertext structure, a hypertext versioning system must provide a visualization of the difference between the two networks being merged, such as the Graph-Unification-Merger and Graph-Comparison-Merger proposals in [90].

Interactions with other goals. Visualizations are often employed in graphical user interfaces, and hence affect how a user interacts with a system (Section 7.11). As noted above, merge support adds additional visualization requirements (Section 7.6). Visualizations can also provide information used when navigating through an versioned space (Section 7.7). Abstractions introduced while satisfying any of the requirements often end up requiring changes to existing visualizations, and the introduction of new visualizations.

7.10 Traceability

Versioning supports reuse of work parts in other works. A hypertext versioning system should provide the ability to trace the reuse of material from an original source to derivative works, and vice-versa, even when it spans work boundaries [87], p. 45. This addresses the problem of losing the object identity, identified in [89], which occurs when an object is copied into a new composite.

[87] identifies a different kind of traceability goal – the ability to trace a change request described in a work annotation to the modification that fulfills the request, thereby using annotations as a change request management (bug tracking) system. Instead of entering the change request into a separate system, the work (code) is annotated directly where the change needs to be made, providing the advantage that the change request is presented along with its context.

Interactions with other goals. Reuse of work parts is affected by the mutability of work revisions (Section 7.1.3). Searching facilities can be employed as a traceability mechanism (Section 7.8). Use tracing can be used to provide awareness among collaborators (Section 7.6).

7.11 Goals for User Interaction

Most authors can cognitively handle the concept of multiple revisions of individual documents. However, since hypertext is a new phenomenon, people who create hypertexts do not have years of authoring experience to draw upon when mentally visualizing a hypertext network as it changes over time. Furthermore, due to the presence of links, hypertexts are more complex than individual documents, and having multiple states of the hypertext over time piles on more. As a result, many sources have stressed the importance of reducing the user-visible complexity of hypertext versioning systems [87,141,154,200]. Østerbye is the most strident on this point, identifying naming of new works and links (echoing Conklin [33]), and version selection as two significant sources of cognitive overhead, addressed in HyperPro by the use of contexts [141]. Another source of complexity occurs when authors have to decide explicitly for every update whether it should result in a new revision [87].

Interactions with other goals. Judicious use of visualizations can help alleviate the user-perceived complexity of hypertext versioning systems (Section 7.9). In open hypermedia systems, tools integrated with the system are responsible for much of the user interface, and hence tool integrations directly affect the user's interactions with hypertext versioning facilities (Section 7.12). Overall, the fewer features a hypertext versioning system provides, the lower the user perceived complexity.

7.12 Goals for Tool Interaction

For VerSE [91], the Hypermedia Version Control Framework [100], the Delta-V protocol [199], Hyperform [202], and the Chimera versioning proposal [200], a major goal is to provide a versioning-aware functionality layer that can be employed by multiple tools. However, these tools must either be coded from scratch to use the version-aware hypertext infrastructure, or existing third-party applications must be integrated, a well-known issue for open hypermedia systems [44,198]. The Achilles heel of infrastructure development is that, in order to be relevant, the infrastructure must be used. This certainly applies to version-aware hypermedia infrastructures, which are useless unless applications employ their functionality. To address this concern, the goals for tool interaction espoused by these systems address how to make the infrastructure as attractive as possible so a tool integrator will mate their tool with the system.

Since application integration is often time-consuming, it creates a non-trivial barrier preventing integration. Not surprisingly, reducing the infrastructure's demands on third-party applications [200], and aiming to maximize the cost/benefit ratio of integrations [91] p. 226, are two goals explicitly named in the literature. Furthermore, since versioning *unaware* systems frequently extended with versioning capabilities, it is a goal to ensure that pre-existing tool integrations must still work once versioning services are added, so as not to destroy the investment in the existing tool integrations [100,199]. The version-aware infrastructure must maintain downward compatibility, and hence versioning aware and unaware applications must be able to interoperate.

However, reducing the barrier to entry for integrations doesn't address the issue of whether the infrastructure services are useful for a particular application. A common goal is that the versioning services should be applicable to as wide a range of tools, application domains, and versioning styles as possible [100,91]. That is, ideally it should be impossible to reject a versioning-aware infrastructure due to lack of functionality, or a mismatch in applicability. Hicks, in [100], identifies three specific qualities to meet this goal:

- **Flexibility:** "to support a variety of different development methodologies, version control services should be flexible, unconstrained by any specific development paradigm." [100], p. 129.
- **Extensibility:** since the complete range of third-party applications is unknown in advance, the version-aware infrastructure should be able to accommodate new types of applications.
- **Scalability:** since applications will vary in their use of versioning services, the infrastructure should be capable of handling both heavy, and light usages.

Several practical goals emerge once the Internet is the communications layer used to access the versioning infrastructure. Authentication is a concern, so that the versioning services can trust the accessing application, and vice-versa. Since it is desirable to remotely collaborate with individuals who should have access to a particular hyperdocument, but who should not have login privileges on a remote system, these should be decoupled [161]. That is, there is a need to give external collaborators write access, without giving them all the privileges of local users. Since, on the Internet, documents are being sent across a public network, if the contents of these documents must be kept private, some mechanism for encrypting their transmission must be used [199]. Furthermore, the Internet raises the problem that people from multiple countries, speaking a multitude of natural languages will be interacting with the versioning infrastructure. In this case, human-readable fields, such as attribute values, version labels, etc., will require internationalization [199]. Finally, since versioning requires the use of time, it is necessary to ensure that the clocks on interacting tools are synchronized [54].

Interactions with other goals. Typically, every new feature added to a hypertext versioning system increases the size of the system's programmatic interface (API), and increases the scope of application integrations. Thus, requirements that typically lead to the introduction of new abstractions, such as revision history support (Sections 7.1.1 and 7.4.1), change aggregation support (Section 7.3), structure versioning (Section 7.4.2), and workspaces (Section 7.6), usually increase the complexity of tool interactions. Authentication of the users of a system supports collaboration by reliably identifying individuals (Section 7.6).

7.13 Goals for Interactions with an External Repository

Most hypertext versioning systems provide versioning services within the system. The system itself contains the code used to store work revisions, retrieve older revisions, store version histories, etc. However, this duplicates the functionality available in existing versioning and configuration management systems. Furthermore, there are some use environments, such as existing software development projects, where all documents are under the control of an existing configuration management system. In these situations, it is unlikely that the benefits offered by hypertext will overcome the cost of switching to a new repository, especially when, as is the case with current hypertext versioning systems, they do not offer the same functionality. In this case, in order to provide hypertext versioning functionality, the existing versioned repository must be used [38], p. 27. This scenario motivates the first goal in [200], which states that "objects stored external to the hypertext system and hypertext structures stored internal to the hypertext system must be capable of independent development." (p. 46). This raises the problem of synchronizing the versioned works in the external repository with the versioned structure within the link server.

Interactions with other goals. When an external repository assumes responsibility for versioning works, it has a broad effect on how other requirements are met. Responsibility for meeting some requirements ends up split between the hypertext system and the external repository; such is the case for revision mutability (Section 7.1.2), co-existence of versioned and unversioned objects (Section 7.1.3), human readable names (Section 7.1.5), linking to a specific revision (Section 7.4.3), collaboration support (Section 7.6), searching (Section 7.8), and visualization of the versioned space (Section 7.9). Other requirements shift to being the primary responsibility of the external repository, including versioning of works (Section 7.1.1), versioning for all content types (Section 7.1.4), reference stability (Section 7.2), change aggregation (Section 7.3), and variant support (Section 7.5). Since applications can continue to use existing repositories, support for this requirement reduces data integration effort (Section 7.12).

7.14 Namespace Interactions

Hypertext versioning systems differ in whether the objects being versioned are named, in addition to having some form of object identifier. For example, in the Chimera versioning proposal [200] works (called objects in Chimera) are files, and hence have filenames, while in the Delta-V protocol [199], Simonson and Berleant et alli [173], and Pettengill and Arango [150], works are named with URLs. In the context of the Web, the Delta-V protocol has identified several goals for adding versioning information to the URL namespace:

- **Each work revision has its own URL.** This ensures that hypertext links can be made to every work revision.
- **Relative URLs should not be disrupted.** If all of the revisions of an object are placed within a subcollection, for example, if all the revisions of “index.html” are placed within a collection with the same name, and hence the first revision would be “index.html/1”, then relative URLs within this resource will no longer refer to the correct destination.
- **Requests on the URL for a versioned object should return a default revision.** If there is a URL for a versioned object, then requests on that resource should be redirected to a specific revision.

Additionally, Sommerville et al. identify the goal that placing a Web page under version control should not affect future accesses to the page’s original URL [180].

Interactions with other goals. Each revision having a separate identifier is a prerequisite for allowing links to a specific revision (Section 7.4.3). Since revision selection rules are often used to make a versioned object return a default revision, this implies searching support (Section 7.8).

7.15 Maintaining Consistent Structures

One goal that is not present in any of the surveyed systems is the ability to revert to a prior revision of just a single work within a hypertext, having all the links adjust to maintain consistency. The ability to pick a specific individual revision of a work independent of the other revisions in a container is a common feature of software configuration management systems. Hypertext versioning systems require that reverting to a previous revision of a work involves reversion to a previous composite revision containing the desired work revision. This is undoubtedly simpler, since it provides a reasonable answer to how best to revert the links that begin or end at a single reverted work. But, the prevalence of this operation in configuration management systems suggests that hypertext versioning support for software development will need to support it too.

Interactions with other goals. If an external repository holds work revisions, it increases the likelihood that a single work object will be reverted (Section 7.13). Consistency is often maintained within the context of a composite, or other structure container that is being versioned (Section 7.4.2).

Chapter 8

Design Spaces and Tradeoffs Associated with Domain Requirements

To satisfy a particular domain requirement, it is necessary to either introduce new elements into a system's data model, add new constraints to the existing system, or both. Usually there are many possible choices for satisfying a particular requirement, creating a design space, where it is necessary to choose a point in this space after evaluating the tradeoffs. For each domain requirement, this chapter describes the particular data model additions, constraints, and design spaces that can be used to satisfy it. Two sections in this chapter do not correspond to any domain requirement. Section 8.1 describes the design space for object organization, which is foundational since most other design spaces involve interactions with objects. Since Section 2.1 defines an object as the representation of abstraction, the need to choose an object organization is driven by the needs of representation, not to satisfy a particular requirement. Section 8.2.2 on versioning of dynamic work content and dynamic links was added to provide some discussion of its difficulties. Versioning in the face of dynamism has not been addressed by existing hypertext versioning systems, and hence there are no domain requirements in this area. However, the remaining sections correlate closely to the domain requirements described in Chapter 7.

8.1 Object Organization

The objects used to represent major abstractions in hypertext, document management, and configuration management systems, fall into one of three broad organizations: all data, data plus properties, or all properties. Unix files are an example of the *all data* type, since a Unix file only consists of a sequence of bytes, and has no associated metadata. If metadata is added to an all data object, there is a main chunk of data, which is the primary piece of state, and multiple properties, typically attribute-value pairs, that are secondary to the main state. The object model in Dexter [95], Neptune [45], and WebDAV [201] are examples of the *data plus properties* organization. In the *all properties* organization, there is no privileged piece of state – all state is a property, including the state that would be considered the main chunk of data in the other two organizations. Aquanet [126], CoVer [87], Neptune/HAM [46], and HyperPro [141] are examples of the all properties organization.

Once an abstraction's explicit representation has been decided, the next concern is which elements of its organization have identifiers. Invariably there is an identifier for the object as a whole, however, there are frequently identifiers for each property too. Typically these identifiers are subsidiary to the main identifier for the abstraction, such as "the author property on work #595". However, it is possible for metadata to have identifiers that directly refer to a specific instance of a property and which are independent of the main abstraction identifier. It is also possible to have multiple identifier spaces within the set of properties, such as in the Dexter model, where some properties store metadata, and others store anchor information.

8.2 Data Versioning

8.2.1 Persistent Storage of Revision Histories

Objects are used to represent such abstractions as works, anchors, links, containers, workspaces, etc., within the computer. This section presents the *revision history design space*, describing several different techniques for recording the revision history of objects. The revision history design space is foundational, since it describes the basic techniques for recording the evolution over time of works, anchors, links, containers, and workspaces. These approaches create an archive of past revisions, and add the dimension of time where there was previously only the current time. The design space of structure versioning (Section 8.5.2) directly builds on this design space, since it depends on exactly how works, anchors, and links are versioned.

There are several approaches for recording the revision history of an object.

State-based approaches:

- **Versioned objects:** In this approach, each revision is a separate object that is *referentially* contained within a versioned object, a container that holds all revisions of the object. Links or relationships record the predecessors and successors of revision objects. The containment relationship between the revision objects and the versioned object must be a reference type (as discussed in Section 4.2), typically a containment relationship on the container object. An advantage of this approach is the ability to record metadata both on the versioned object and on the individual revisions.

  There is also a range of choices for how to represent the predecessor/successor relationships. Since each revision can be viewed as having a set of predecessors, and a set of successors, the containment design space is applicable. Since the versioned object approach requires each revision to be a separate object, an individual revision cannot use inclusion containment for its predecessors and successors. However, all of the referential containment types could be employed. It is possible for each revision to store the predecessor/successor relationships on the revision, an example of the "container holds containment relation" type from Section 4.2. It is also possible for revisions to hold only the predecessor relationships, wherein the predecessor relationships on child revisions do double duty as the successor relationships for its parent. First class relationship objects, or hypertext links, can also be used (see Figure 21b), and have the advantage that the revision objects do not need to store predecessor/successor information, and could potentially participate in multiple version histories. First class relationship objects must themselves be contained within the versioned object, and any reference containment type can be used for this. Of course, if inclusion containment is used, the relationship loses its first-class status since it is no longer an independent object.

  When revisions are contained using multiple containment, the revisions can belong to containers other than the versioned object, such as user-created containers (e.g., folders or directories), workspaces, and configurations. When the container contents themselves are versioned, multiple containment allows revisions to be reused across revisions of these versioned containers, thus resulting in fewer objects than would occur if this reuse wasn't possible, and the revisions needed to be copied to belong to each container revision. Referential containment of revisions, in conjunction with a decentralized name or address space, permits version histories that span organizational or machine boundaries by having the versioned object contain revisions that are on multiple machines, potentially machines owned by different organizations. This has the drawback of either needing some mechanism for maintaining the integrity of the references, or accepting that there could be references that are not up-to-date (e.g., if a revision is deleted).

Systems that contain revisions inside versioned objects using reference containment include HyperPro [141], CoVer [88], HyperProp [178], and the Hypermedia Version Control Framework [100].

- **Within-object versioning:** In this scheme, the versioned object uses *inclusion* containment to hold revisions, and hence all revisions are *within* the versioned object. Thus, *within-object versioning differs from the versioned object approach in the type of containment used, within-object versioning employing inclusion, and versioned object using referential types.* Within-object versioning is shown in Figure 21d. Examples include the ",v" files of RCS [185] and the ".s" files of SCCS [163], the versioning capability of some word processors (e.g., Microsoft Word), along with Palimpsest [56], VTML [194,18], EH [74], P-Edit [111], MVPE [166], Historian [1], VE [11], Timewarp [57], and Delta [37]. While these systems all share the quality of inclusively containing all revisions, their concrete representations vary significantly.

  The advantage of this technique is that all revisions are stored within a single object, and it is possible to guarantee the stability of references within these objects, since the current location of an endpoint can always be computed. When changes are recorded down to the keystroke level, within-object versioning can support remote collaborative authoring where all collaborators simultaneously work on the document, since all operations by all collaborators are recorded. Recording all revisions can be a drawback, since a publicly accessible document might not want to publicly show all of its prior revisions.

  Within-object versioning has the drawback that revisions cannot participate in other containment structures, unless a replica of a specific revision is made and then placed into the container. Alternately, it is possible to use multiple containment to hold the entire versioned object, and its included revisions, within multiple containers.

Two significant design choices for within-object revision are the size of the minimum length content chunk, and the range of attributes that can be set on each chunk. Chunk size varies, with the largest minimum chunk size being a programming language function [1], but with other choices being a single line, as with the C preprocessor, a programming language token [37], all the way on down to a single character [56,194]. Settable attributes always include the person who made a change and the time when the change was made. Other common attributes include the revision number of the change, and a rationale/comment field. Most systems limit settable attributes to those that are predefined by the system, however some provide the ability to set arbitrary attributes, and retrieve them using predicates. Arbitrarily settable attributes allow within-object versioning systems to also handle within-object variant representation tasks, discussed further in Section 8.6.1.

- **Predecessor/successor relationships only:** Each revision is stored in a separate object, but no container object represents a particular versioned object (see Figure 21e). Typically a repository, or super-container holds all revisions of all objects, as well as all relationships between them, in a large pool of objects. The advantage of this approach is that it can support versioning without using container objects. In conjunction with a decentralized name or addressing scheme, it can also model revision histories that span organizational and machine boundaries, since it avoids the issue of which machine hosts the collection representing a versioned object. However, when revision histories span organizational boundaries, referential integrity is a potential problem, as communication and coordination between the machines storing the individual revisions cannot be guaranteed. Disadvantages of the approach include inefficient revision selection, hence inefficient creation of arbitrary configurations, and inefficient setting of metadata that must be unique across the version history of an object, such as labels. Examples of this approach include Xanadu [137], and the NTT Labs. versioning proposal [143].

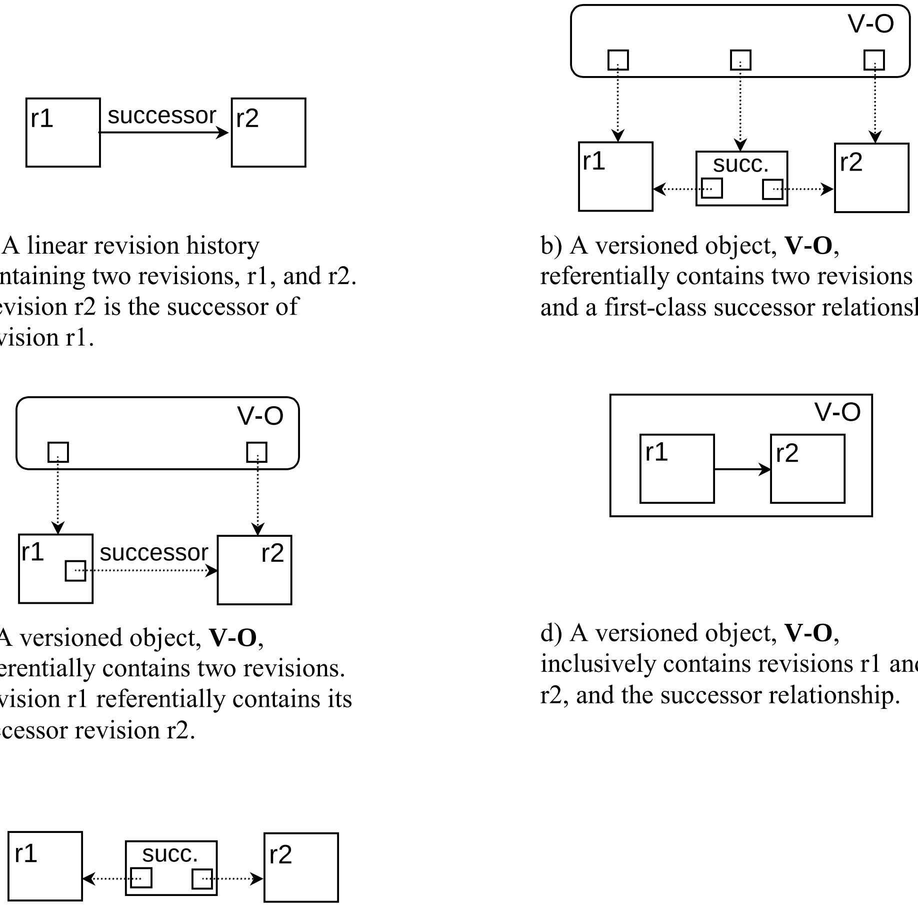
State-based approaches for recording a revision history are summarized in two figures below. Figure 21 shows how the state-based approaches are used to represent a two-object revision history, and provides instances of each of the objects used to represent the example revision history. Figure 22 also shows the revision history design space, but does so using entity-relationship diagrams [29] showing a revision history at three layers of abstraction, using an approach similar to that used for containers in Figure 7. In the abstract relationship layer, a revision history is modeled as a revision with a predecessor relationship, reflecting that a specific instance of a revision history is a set of revisions connected by predecessor relationships. The explicit relationship layer shows the versioned object, within-object, and predecessor/successor relationships approaches, highlighting that these three approaches all represent the abstract revision history in the layer above. At the concrete representation level, for each of the approaches in the explicit relationship level, one or more examples of concretely representing the revision history are shown. Representations shown in the concrete representation level are capable of, at minimum, representing the objects and relationships shown in explicit relationship level. They may be capable of more than this. VTML and Palimpsest, for example, are capable of also representing fine-grain changes, and could form the concrete representation for a change-based approach as well.

The advantage of the three-layer model for revision histories is its separation of the abstract notion of revision history, shared by all state-based approaches, from the high-level overview of each versioning approach (versioned object, within-object versioning, predecessor/successor), which is in turn distinct from its specific concrete representation. It allows the characteristics of each versioning approach to be considered independent of the idiosyncrasies of its specific concrete representation, and firmly places delta storage concerns in the concrete representation, where it has no impact on higher-level modeling of the revision history. It also highlights similarities between approaches that otherwise seem quite different, placing systems such as RCS and VTML in the same category of within-object versioning systems, since they share modeling similarities at the explicit relationship layer, but have significant differences in their concrete representations.

Change-based approaches:

- **Layers:** A composite object is created that holds a subset of system objects. Typically the set of changes represents a logical change, or variant, of the system with respect to some checkpoint. This approach differs from versioned objects in that these composites may contain one revision of several different objects, while a versioned object contains many revisions of the same object.

Combining layers creates system revisions, with rules used to evaluate which layer is chosen in the case of conflicting changes. If the composite contains its objects by-value, each object can only belong to one layer. This had the advantage of making it easier for people to understand how a particular change ended up being selected, since each object only appears in one composite. By-reference containment is also possible, and has the advantage that a single change can be stored in multiple layers, making it easier to create system variants using part of, but not all of another change. An example of this approach is the PIE system [80].



a) A linear revision history containing two revisions, r1, and r2. Revision r2 is the successor of revision r1.

b) A versioned object, **V-O**, referentially contains two revisions and a first-class successor relationship.

c) A versioned object, **V-O**, referentially contains two revisions. Revision r1 referentially contains its successor revision r2.

d) A versioned object, **V-O**, inclusively contains revisions r1 and r2, and the successor relationship.

e) Revisions r1 and r2 are independent objects, connected by a first-class link that represents the successor relationship.

Figure 21 – State-based design choices for representing a linear version history.

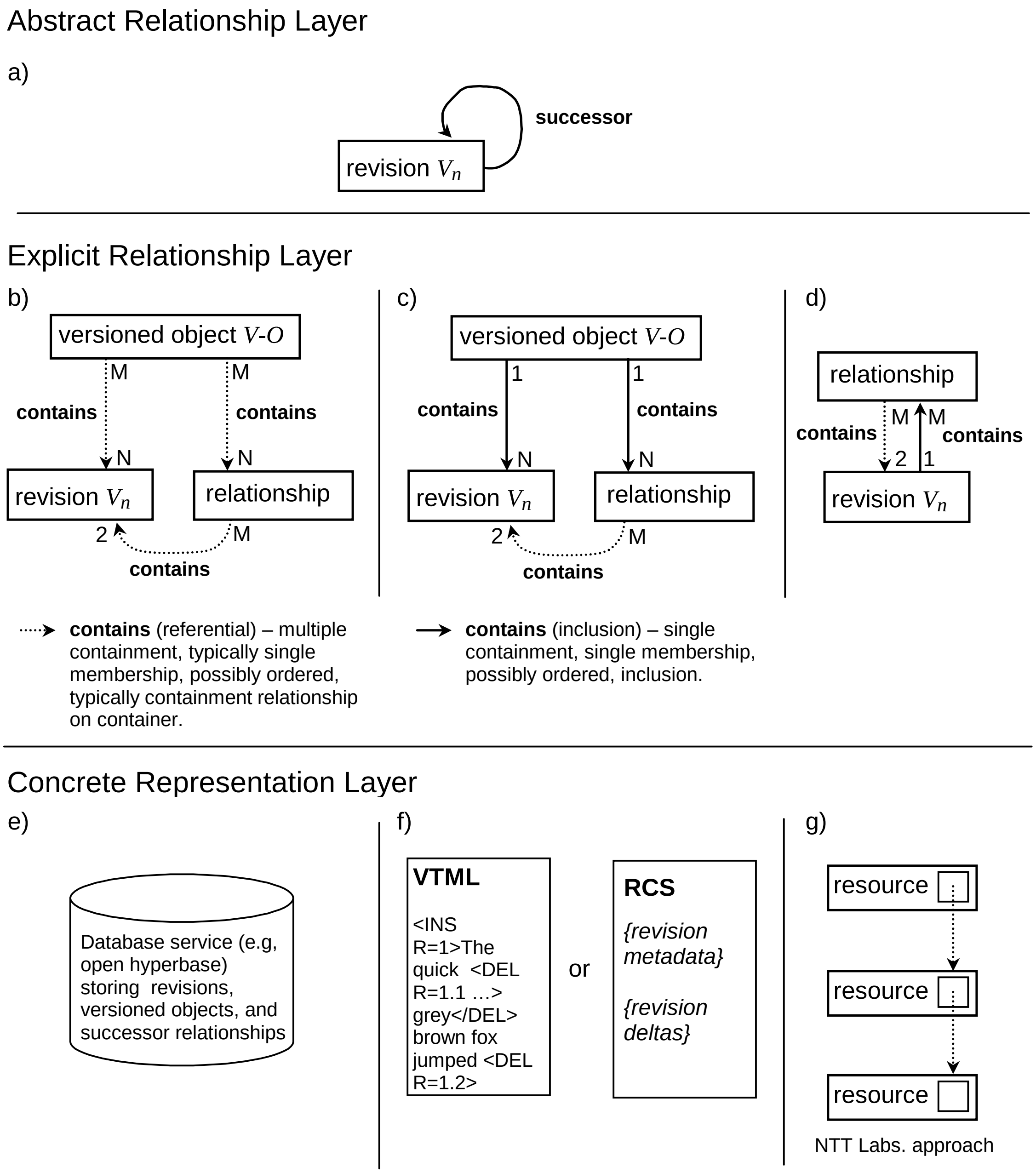

Abstract Relationship Layer
a)
successor
revision V_n
Explicit Relationship Layer
b)
versioned object V-O
M
M
contains
contains
N
N
revision V_n
relationship
2
M
contains
c)
versioned object V-O
1
1
contains
contains
N
N
revision V_n
relationship
2
M
contains
d)
relationship
contains
M
M
contains
2
1
revision V_n
contains (referential) – multiple containment, typically single membership, possibly ordered, typically containment relationship on container.
contains (inclusion) – single containment, single membership, possibly ordered, inclusion.
Concrete Representation Layer
e)
Database service (e.g, open hyperbase) storing revisions, versioned objects, and successor relationships
f)
VTML
<INS R=1>The quick <DEL R=1.1 …> grey</DEL> brown fox jumped <DEL R=1.2>
or
RCS
{revision metadata}
{revision deltas}
g)
resource
resource
resource
NTT Labs. approach

Figure 22 – The state-based versioning design space. In (a), the abstract relationship layer depicts a revision history as a series of revisions with predecessor relationships. In the explicit relationship layer, three possible design choices are shown: (b) the versioned object approach, (c) within-object versioning, and (d) predecessor/successor relationships only. In the concrete representation layer (e-g), a non-exhaustive set of examples is given of possible representations of the objects and relationships in the explicit relationship level.

	Approach	Advantages	Disadvantages	Figure	Examples
State-based	versioned object	Can record metadata on versioned object and revisions. Revisions can participate in multiple containers, such as workspaces, configurations, etc. Reduces object duplication when versioning parent containers	Requires multiple objects to represent a version history.	Figure 21b, Figure 21c, Figure 22b	HyperPro [141], CoVer [88], HyperProp [178], Hypermedia Version Control Framework [100]
	within-object	All revisions stored within a single object. Reference stability within revisions can be guaranteed. Can efficiently track fine-grain changes.	Revisions cannot participate in multiple containers, unless replicas are used. All prior revisions are accessible, limiting confidentiality.	Figure 21d, Figure 22c	RCS [185], Palimpsest [56], VTML [194], P-Edit [111]
	predecessor/ successor links only	Revision history can be represented without using a container. Can model version histories that span organizational and machine boundaries.	Inefficient revision selection, inefficient creation of configurations. Inefficient setting of metadata, such as labels, that must be unique across all revisions. Referential integrity of version history can be a problem.	Figure 21e, Figure 22d	Xanadu [137], NTT Labs. versioning proposal [143]
Change-based	layers	Revisions are represented as the combination of a set of logical changes. Good support for representing variants.	Inconsistency can arise between mutually incompatible changes. Difficult to map into hierarchical name systems (like filesystems).		PIE [80]

Table 6 – Design options for recording the history of objects, including representative, but not exhaustive examples of each.

8.2.2 Versioning Dynamic Content and Dynamic Links

Hypertext systems frequently have dynamic content where the symbolic representation of work is the output of a computational process, as well as dynamic links, where the endpoints of the link are computed. For example, on the Web, dynamic content can be the result of Java [81] or JavaScript [63] programs executed within the Web browser, or the result of a computation executed on the server. Server-side dynamism often involves displaying the results of a database query, as exemplified by search engine, airline ticket reservation, and stock quote sites. Microcosm [43], with its link filter architecture, is an example of dynamic links: link traversals pass through a series of filters, with each filter determining whether it contributes endpoints for the link.

Versioning dynamic content, and dynamic links is hard. Dynamism involves running a program, with its explicit and implicit dependencies on its environment. Operating system version, memory requirements, processor requirements, network connectivity, other programs, such as databases, that must also be available in the environment — all are possible external dependencies of dynamic content and links. Many of these factors are outside the control of the hypertext versioning system. Hardware version and operating system version can be checked by a version control system, but if they are incompatible, a version control system can do little more than flag the incompatibility for an operator to handle. While it is not unheard of for a site to place every single file under version control, this still does not address the entire problem, since the controlled files have dependencies on the hardware and network environment, and these are outside the scope of control of a version control system.

Most approaches to versioning dynamic content reduce dynamic content to some form of static state, such as a description of the process in the form of a program, or executable image. A snapshot can be taken of this static state, and versioned, possibly along with those aspects of the environment that affect its execution, and that are capable of being versioned, such as environment configuration files. One goal of this approach to dynamic content versioning is to be able to revert, at some future time, to a stored state of the dynamic process, and then be able to re-execute that process.

The Web, with its client/server split, suggests some additional ways of versioning dynamic Web pages. If the goal of versioning is to archive the content, instead of the process that generated the content, representative snapshots of the Web pages as visible at the browser could be versioned. Thus, instead of versioning the process that generates a graph of a stock's value during the day, take a snapshot of that page at specific instants during a day. For a stock quote, it might be adequate to take a snapshot when the stock market opens, at mid-morning, noon, mid-afternoon, and at market's close. Or, for versioning a search engine's pages, the top thousand queries could be executed once a week, and their results stored. Periodic intelligent traversal of many sites, from auctions to news feed sites, could allow their contents to be versioned.

Java and JavaScript, since they run on virtual machines within the Web browser, provide the ability to version both the operating environment, the browser's configuration, and also the "hardware," the virtual machine and the browser executable itself. The use of a virtual machine places the "hardware" under the control of the version control system. Since JavaScript is a standard, and there is a relatively small number of platforms (Web browsers) on which it runs, there is a greater chance that, if a JavaScript program and Web browser were placed under version control now, several years from now it will still be possible to run that program. Nevertheless, the ability to version the virtual machine still does not address external dependencies such as network connectivity, and services that are under the control of another organization, such as a database, or news feed.

8.2.3 Revision Mutability, and Immutability

This requirement adds modification constraints to the data model used for representing the history of objects. As discussed in Section 8.1, an object can have one of three organizations: all data, data plus properties, and all properties. For the following discussion, the term *primary state* describes the data item that represents the work within an object, distinguished from the object's *metadata*, which is all other non-work object state. The three choices for the mutability of objects are: primary state and metadata are immutable, primary state is immutable while metadata is mutable, and both primary state and metadata are mutable (i.e., the entire object is mutable). The case where the primary state is mutable while metadata is immutable is considered to be a degenerate case, since metadata contains assertions about the work, and hence if the primary state can be modified, the assertions must be capable of change so they can be kept correct.

- **Immutable primary state and metadata:** This adds the constraint that, once the primary state is no longer being actively worked on (e.g., once it has been checked-in), neither it, or its metadata can be modified. This has the advantage of faithfully recording all aspects of an object at the time it was frozen, providing the best capture of the past state. It has the disadvantage that if access control is stored as metadata, the access permissions cannot be changed once the object is frozen. Since this is undesirable, it implies that access control directives must be separated from the object.
- **Immutable primary state, mutable metadata:** This adds a constraint that some, or all metadata can be modified on each revision. Mutable metadata may affect access control, if access permissions are stored on each revision. This has the advantage of allowing access permissions to be flexibly modified at a later date, while still keeping access permissions stored with the object they affect. Anchoring and linking may also be affected, if anchors and links are represented as metadata. This has the advantage of allowing new links and anchors to be created to frozen items, allowing annotations, and new relationships to be explored.
- **Mutable primary state and metadata:** In this case, the primary state and the metadata can be modified on all revisions, even if they have been checked-in. This adds the constraint that reliable configurations cannot be created if the primary state has been modified. Since a configuration is a collection of individual revisions, if one of these revisions is modified at a later date, then it is not possible to recreate a specific configuration.

8.2.4 Coexistence of Versioned and Unversioned Objects

By “coexist,” this goal implies that the objects coexist within the same container, and typically this issue only applies to containers employed by the end user to group their objects. This goal adds a constraint (or lack of a constraint) on what items a container can contain. The set of all possible constraints includes:

- **Versioned objects only:** Only versioned objects can be contained. This constraint prevents the containment of unversioned objects.

- **Revision from a versioned object only:** Only a specific revision from a versioned object may be contained. This is accomplished by adding a revision selection rule to the containment relationship. This constraint prevents the containment of unversioned objects.
- **Unversioned objects only:** Containers can only contain objects that are not under version control (i.e., that are not versioned objects, or that are not revisions of a versioned object).

To meet the goal of allowing both versioned and non-versioned objects to coexist, *none* of the above constraints can hold.

This goal only applies to state-based versioning, where each revision is a separate object. For within-object versioning, this does not apply, since the container does not have to distinguish between unversioned documents, and within-object versioned documents: both appear to just be objects for the purpose of containment.

Allowing containment of both versioned and unversioned documents increases system complexity, since containers (either composites, or directories in a hierarchically organized namespace) now have to handle the issue of whether a versioned container should record the membership of an unversioned item when its state is frozen. If it does record the membership, it is possible that, in the future, the unversioned items will be deleted, and the prior container state will now contain dangling membership links. If, on the other hand, the unversioned items are not recorded as part of the container's state when it is frozen, containers are now complicated with two kinds of membership, versioned and unversioned. Furthermore, when a versioned container is reverted to a prior state, the system is faced with the choice of either removing all unversioned items from the container, or preserving the unversioned items and only reverting the versioned items, making it so that the exact prior state of a container can never be recovered. Due to these complexities, hypertext versioning systems that version containers adopt an all or nothing approach, requiring that all of a versioned container's items be versioned.

8.2.5 Versioning of all Content Types

The goal of versioning all content types discourages versioning approaches that are dependent on knowledge of the internal organization of a document. For example, while within-object versioning is certainly possible for all document types, to date it has primarily been applied to text. Hence, if support for a wide range of document types is needed, either the within-object versioning technique will not be usable, or it must be adapted for all the needed document types. Fine-grain change tracking, such as that provided by Palimpsest [56] and VTML [194], requires modifications to programs that operate on a specific data type, and hence supporting within-object versioning implies changing these programs. If there is access to source code for the applications, this is a possible, but major, engineering task. Of course, if there is no source code access, adding within-object versioning is impossible. Alternately, it is possible to post-process an object after it is stored to determine what has changed, however, this too requires that post-processors have knowledge of the internal structure of the object.

Similarly, this goal limits the kind of concurrency control mechanisms that can be used. Synchronous collaborative authoring applications such as Grove [58] and Prep [139] willingly forgo content type independence to provide fine-grain merging down to the keystroke level. The approach in Grove and Prep of sending fine-grain update notifications requires a constant network connection to be held open between collaborating applications. This requires such systems to be fully network connected throughout collaborative authoring sessions, and easily reconnect accidentally disconnected authors. The semantics of the update messages depend on the type of document being edited. For example, a spreadsheet requires knowledge of cells and formulas, and a bitmap image editor requires knowledge of colors, brushes, masks, etc. While it is possible to extend this concurrency control technique to many document types, it too would be a significant engineering task both to define the correct event notifications, and to add the network support into applications to receive and process these notifications.

Duplex [144] and Alliance [165] also exploit special knowledge about the internal hierarchical structure of their documents to provide concurrency control on document subtrees. When considering using this technique for all document types, two problems arise. First, not all documents are hierarchically organized. For example, bitmap images are not meaningfully decomposable into trees for the purpose of concurrency control. While the technique could be extended to make use of any reasonable modular decomposition of the document, this would not get around the second problem, that of requiring application programs to be modified to support a standard concurrency control mechanism for each document type. While it is certainly feasible to do this, it would require significant engineering effort to accomplish.

Two concurrency control mechanisms that work well across multiple document types are whole-document locking, and parallel development. These techniques have the advantage that it is possible to create a helper application that manages the concurrency control (taking out a lock, or checking out a document), thus allowing all applications to interact with all document types without needing explicit knowledge of the concurrency control scheme.

8.2.6 Revision Naming

The standard way of providing revisions with human readable names is to associate a name called a *label*, with a specific revision. This is accomplished either by adding a *revsision, label* pair as metadata on the versioned object, or by putting the label as metadata on a specific revision.

Labels are frequently used in revision selection rules, as part of a predicate that identifies a specific revision of a versioned object. For example, the rule "label = 'Beta2'" would select any revision that has a "Beta2" label on it. Often it is desirable to ensure that a maximum of a single revision is returned for a given revision selection rule. To ensure this, a label value must be unique across all revisions of a versioned object. However, this seemingly innocuous requirement ends up discouraging distributed revision histories, due to the difficulty of checking for uniqueness across a distributed revision history tree. Consider a revision history tree that spans three locations. Adding a label to a revision at one location requires a check for uniqueness at the other two locations, thus increasing the time it takes to perform the operation, and raising the possibility that the other locations might not be accessible, thus preventing the operation from taking place.

Evaluating revision selection rules also becomes more difficult for distributed revision histories, since the predicate needs to be evaluated at multiple sites. Like adding a label, evaluating revision selection rules on distributed revision histories is slower, due to the need to evaluate at multiple sites, and is susceptible to interruptions when accessing sites. The reduction in speed has significant impact on configuration management, since configurations are defined using a series of revision selection rules, and hence their efficient evaluation is key for efficient creation of configurations. Since configurations can comprise hundreds, or thousands of objects, every slowdown in rule evaluation is magnified many times.

8.2.7 Removing revisions and versioned objects

The primary design choice affecting deletion of revisions and versioned objects concerns whether the semantics of delete are *destroy* or *mark as deleted*. When an object is destroyed, the resolver function no longer maps its identifier to its persistent storage, and, sometimes, the area in persistent storage that had held the object's contents is overwritten with zeros, ones, or random contents, to prevent any non-trivial reconstruction of the destroyed object's state. In constrast, the *mark as deleted* approach simply sets the "deleted" flag on the object, and does not affect mappings of identifiers to the object. Hybrid approaches are possible. For example, it is possible to mark revisions as deleted, and then destroy these marked revisions by reaping them after a fixed time period.

The destroy approach has the advantage that it frees the storage space used by the object. Thus, long revision histories can be compacted by removing old, unused revisions, and older documents can be removed, and no longer need to be archived. Destruction carries a heavy consistency penalty. If the destroyed object participated in any hypertext links or containment relationships, these are left dangling, and must be repaired. If the link or container is an immutable revision, destruction of a contained object results in irrevocable inconsistency. Destruction of revisions can be complex when within-object versioning is used, since it involves recalculation of the internal state of the object. For example, destruction of a revision in an RCS [185] ",v" file requires recalculation of the reverse deltas inside the file.

The mark as deleted approach avoids the inconsistency problems of destruction. Objects marked as deleted are not shown in most user interfaces and visualizations, and hence appear to the user to be deleted. However, they can be shown when a composite (or other versioned container) is reverted to a previous revision that contains the deleted object. The deleted revision can also be made visible if a link traversal ends at that revision. Though it has the advantage of avoiding the inconsistency problems of destruction, the mark as deleted approach also avoids the advantages of destruction. All the storage space is still used, and deleted objects are still archived.

8.3 Stability of References

There are two main problems that can occur with references such as links [42]. If the endpoint of a reference no longer resolves to an object, then the reference is considered to be *dangling*; such a hypertext link is called a *dangling link*. When the anchor of a hypertext link does not correctly identify the correct content within the referenced object (for example, if the object has been changed, unbeknownst to the hypertext system) it yields the *content reference* problem.

Dangling links occur when the target of a reference or link has been deleted (made inaccessible from the target location in the namespace) or moved (made accessible from another location in the namespace), and the reference or link has not been modified to account for the change.

Versioning is one of several solutions to the content reference problem; others are covered in [42]. The essence of the versioning approach is to ensure the linked-to revision is always available in the form it had when the link was created. Assuming the revision is permanently stored by the system (i.e., isn't deleted), the referenced endpoints will always be available. This approach carries with it some historical assumptions about naming and link traversal semantics.

First is that each revision has a name, address, or retrieval specification that is usable as a link endpoint. For example, if revisions are stored in RCS ",v" files, each revision does not have an individual name in the filesystem, only the versioned object (the ",v" file) does. Thus, in this hypothetical case, if filenames are used as link endpoints, it is not possible to link to a specific revision, since each revision does not have a separate name. A situation similar to this is encountered with Web-based versioning systems [161,150,173,134] that store all revisions of an object in a single file. Since URLs for filesystem-based Web servers are typically mapped directly into filenames, this raises the problem of creating a mechanism for accessing individual revisions. Typically this is accomplished by appending a revision selection rule (e.g., a revision identifier, or a date) or a revision retrieval operation (e.g., a checkout) to the end of the URL.

To guarantee freedom from content reference problems, the versioning approach assumes that links to objects will always return the originally linked revision, since it is only for this revision that the system can guarantee the link is correct. But, linking to a specific revision is only one of several possible link traversal semantics: it is also desirable to link to the most recent revision, or to be given a choice of revision destination. Since future revisions have also been stored, it is possible that the link endpoint could be located in revisions subsequent to the one originally linked, thus supporting other link traversal semantics.

Within-object versioning systems such as Palimpsest [56] and VTML [194] note this as a benefit. However, no full-featured hypertext versioning system exploits this capability.

A drawback to the use of versioning to solve the content reference problem is that versioning alone cannot prevent dangling links. One solution to avoid this drawback is to prohibit delete and move operations, as is done in Xanadu [137]. However, there are many valid reasons for the owner of a document to permanently remove it, and all its prior states. For example, in corporate settings, mergers and acquisitions can make a company name obsolete, product lines can be modified or terminated, in both cases making it desirable for the owner to remove all documents that reference the old company or product names.

If the original owner of an object decides to delete or move it, yet the benefits of versioning for addressing the content reference problem need to be retained, it is necessary to incorporate third party versioned document stores, where the third party has no compunction about keeping around information the owner has deleted or moved. For example, the implication of a third party caching old company or product names is different from the owner preserving this state. The owner has safely removed the obsolete information, while the third-party store has achieved their goal of archiving the older information. For the third party, no endorsement of the older states is implied, but for the owner of the information, the mere fact of making older information available increases its perceived value. The use of third-party version repositories seems especially useful for the Web, with its average document lifespan of approximately 50 days [152]. Even if the original owner of the material has no motivation for saving older revisions, a third party archive service would.

8.4 Change Aggregation Support

For change-based systems, the ability to provide change aggregation support is inherent, since an abstraction representing a set of changes is necessary just to record previous object revisions. Thus, change-based systems automatically meet this requirement. State-based systems, on the other hand, do not automatically provide support for aggregating changes across multiple objects, and hence the purpose of this requirement is to ensure that they do.

There are three main approaches to meeting this requirement: adding a new container that references the revisions involved in a change, adding a new non-container object that references the revisions involved in a change, employ an existing container to refer to revisions involved in a change, or use labels on revisions to denote that the revisions together represent a change. Using a software development example, fixing a specific problem report might involve modifying several software modules. In the container approaches, the container refers to the revisions that were created in the process of repairing the reported problem. Using labels, a specific label value (perhaps the problem report identifier) would be placed on each modified revision.

- **New Container:** A new container object is introduced into the system, with each container instance representing a logical change. The container points by-reference to the revisions modified in the course of creating the logical change. The container also contains, either as metadata, or by-reference to another document, a description of the logical change. The change aggregation containers are separate from other containers used for grouping objects, such as collections, or composite objects.

  Change aggregation containers typically do not belong in a hierarchy of changes, and hence have a flat namespace, one that is separate from the namespace used for grouping objects that are being authored. It is possible to keep the change aggregation and grouping namespaces completely separate, though typically the change aggregation containers are mapped into an unused portion of the namespace. It is also possible to have change aggregation containers participate in the same namespace as other objects being authored. Taking Web-based systems as an example, change aggregation containers could be named using a non-http URL scheme, or can be placed in an unused part of the http URL space, or can be interspersed with authorable resources in their portion of the http URL space.

- **New non-container object:** A new non-container object is introduced into the system that contains either as its content, or as metadata, a list of references to the revisions comprising a logical change. The non-container object can thus be seen to be acting like a container, holding pointers to the revisions by-reference, though not supporting all container semantics. The advantage of this approach is its simplicity. By not using a container object to group changes, all of the semantics of operations on containers do not need to be supported. This can be an advantage when container semantics are complex. For example, if containers support the ability to order their members, this facility will not be used when recording changes. However, this is its disadvantage as well, since it leads to the need to add capabilities for adding and listing revisions. Like change aggregation containers, change aggregation objects can belong either in a separate namespace from authored objects, or can be placed in an unused part of the authored resource namespace, or can be interspersed with authored objects.

- **Reuse Workspace Container:** Composite-based hypertext versioning systems often use containers to represent workspaces, examples being Neptune [46], HyperPro [141], CoVer [88], VerSE [91], and HyperProp [178]. Each workspace contains the objects being worked on by an individual, or a group. In this case, a specific revision of a workspace container can be restricted to represent a logical change, and hence the workspace container can do double-duty as both a workspace and a change aggregator. The advantage to this approach is that it removes the need for a separate container just to hold changes, thus simplifying the system. It also eliminates the need for a mechanism to assign revisions to a change aggregation container. The disadvantage is that it requires all changes to a workspace, from when it was checked out to when it was checked in, to correspond to a single logical change. This policy might be too restrictive for use cases where the visibility of revisions is affected by the check in. For example, it might be desirable to make several interrelated changes before checking in a workspace if the checkin operation causes workspace contents to be visible to other collaborators. The other disadvantage of tying change aggregation to workspace revisions is a single logical change might span multiple revisions of a single object. This can occur if an object is checked-in, and errors are subsequently found, thus necessitating another revision to satisfy the conditions of the logical change. Finally, it can be difficult to identify exactly which revisions in a workspace were changed when an entire workspace revision represents a logical change.

Further complicating this design space is the issue of how to populate change aggregation objects or containers. A simple approach is to have the user manually add revisions to the change aggregator, perhaps using a change-tracking tool. However, this is a tedious operation, susceptible to errors. Change aggregators can automatically have revisions added to them if they are identified during a checkin operation, thus allowing the checkin command to add the revision to the change aggregator in addition to its normal behavior. It is also possible to indirect the identification of the change aggregator. For example, if the change aggregator is associated with a workspace, then a checkout could identify a workspace, and hence identify the change aggregator that will refer to the changes about to be made. This, then, makes the change aggregator approaches seem similar to the container reuse approach. However, they are different, since the changes are captured due to individual revisions being worked on, instead of the entire workspace being captured as a change.

8.5 Link and Structure Versioning

8.5.1 Link versioning.

The representation of links typically takes one of two forms: either the link is an independent object, or the link is contained within an object representing a work. Using the notation from Section 2.2.1.1, independent links are those where $D(t)$ and $D_l\ (t)$, as well as $P_c(t)$ and $P_{c,l}(t)$, are separate, independent objects, while dependent links are those where $D(t)$ inclusively contains $D_l(t)$, and $P_c(t)$ inclusively contains $P_{c,l}\ (t)$. Independent links have the entire object versioning design space available for representing the revision history of a link. When the link is contained within the object representing a work, it has greater constraints on how it can be versioned; typically link versioning is a side effect of work versioning.

- **Independent links:** As an independent system object, a link's history can be recorded using any of the techniques for recording the revision history of objects, as described in Section 8.2.1. That is, the versioned object, within-object versioning, and predecessor/successor relationship approaches (shown in Figure 21 and Figure 22) could potentially be used.

  For links, the versioned object approach is typically used, wherein a container object referentially contains all revisions of the link. The versioned object approach has the advantage that it permits the creation of composites containing a consistent set of documents and links, such as the most recent revision as of a specific time, or a specific snapshot in the development of the composite. By-reference containment also allows the creation of containers that model a link structure by containing a single revision of multiple links, in this way capturing a link structure [100].

Within-object versioning has the advantage of only needing one object to record all revisions of a single link. It has the drawback of making composite creation more difficult, since either the entire versioned object would need to be contained, or the individual revisions would need to be copied out of the versioned object and placed into the container. As noted in Section 8.2.1, an important design choice for within-object versioning is the size of the minimum length content chunk. This is less important for link versioning. Fine-grain change tracking, as provided by VTML [194] and Palimpsest [56], typically does not provide much value for links, since they have minimal content beyond the link endpoints, and hence do not justify the added complexity of such change tracking. However, if a link has significant chunks of metadata, such as an annotation, fine-grain change tracking of these textual metadata items could be valuable.

Using only predecessor and successor relationships to capture the revision history of links is also possible. This would eliminate the need for a container object representing all revisions of the link, and would permit link revisions to more easily span control boundaries. It has the drawback that it is difficult to efficiently evaluate revision selection rules. No existing hypertext system versions its links in this way.

- **Links a dependent part of works:** In some hypertext systems, links are contained within, and hence dependent upon, the objects representing linked works. Links can be embedded within that portion of the work object representing the content of the work, as is the case with HTML links on the Web. Alternately, links can be contained as metadata about the work object content, as is the case with the source link in WebDAV [79].

  When links are contained within a work object's content (that is, contained within that part of *D(t)* or $P_c(t)$ that represents the content of the work), their history is the same as the content, and hence when the work object content has a new revision made, so too do the links in the content. The versioning of links is completely subsidiary to the versioning of the content. When links are contained as metadata (contained within the portion of *D(t)* or $P_c(t)$ that represents metadata about the work), there are two choices. First, if the metadata is versioned along with the rest of the object, then link versioning is again subsidiary to versioning of the object. However, it is conceivable that metadata could be versioned separately from the main object, with each item of metadata possessing its own revision history. This has the advantage that metadata items, like links, can conceptually be part of the object, but still have independent version histories. The drawback is that this makes the object substantially more complex, since each item of metadata can contain multiple revisions. No existing hypertext system provides metadata versioning services.

The design choices for versioning links are captured in Table 7 below.

	Approach	Advantages	Disadvantages	Examples
Independent links	Versioned object	Permits creation of composites of documents and links. Permits creation of composites of multiple links to represent structure. Permits efficient evaluation of revision selection criteria.	Need many data items to represent revision history.	CoVer [88], VerSE [91], Hypermedia Version Control Framework [100]
	Within-object versioning	One object records all link revisions. Useful if link has large metadata chunks, such as annotation text.	Cannot create composites containing link without replication. Fine-grain tracking of changes (such as annotation text) adds complexity.	None.
	Predecessor and successor relationships	No container object needed to represent link revisions. Can represent link revision histories that span control boundaries.	Inefficient setting of metadata, such as labels, that must be unique across all revisions. Expensive to evaluate revision selection criteria. Maintaining integrity of revision history may be difficult.	None.
Dependent links	Embedded within object	Simplicity: when a revision is made of the object, a revision is also made of the link. When object is deleted, all links are automatically deleted too.	Cannot independently version links. Cannot version structure separate from linked objects.	Web-based: BSCW [19], WWRC [161], WWCM [104], MKS Web Integrity [134], WebRC [75], also [150], [173]
	As subsidiary metadata	Simplicity: when a revision is made of the object, a revision is also made of the link. When object is deleted, all links are automatically deleted too.	Cannot independently version links. Cannot version structure separate from linked objects.	Xanadu [137]
	As independently versioned metadata	Links can be versioned separate from object contents. When object is deleted, all links are automatically deleted too.	Adds significant complexity to the object, since each item of metadata is independently versioned.	None.

Table 7 – Design options for recording link history.

8.5.2 Structure versioning.

A *link structure* is a set of links, and the term *structure* is used in an evocative sense, to describe the graph created by this link set. Abstractly, structure versioning is the act of maintaining the revision history of a link set. Since a set is represented within the computer by a container, the essence of structure versioning is placing a set of links into a container, termed the *structure container*, and then versioning the structure container. This premise underlies the structure versioning design space. The existence of the structure container means the containment design space (Section 4.2) will be brought to bear, and the need to version the structure container brings in the versioning design space (Section 8.2.1) as well. The structure versioning design space thus depends on the existence of these other two design spaces for the terms used to describe its own design choices.

Two criteria determine whether a particular structure versioning design choice is complete. The *symbolic rendition criterion* asserts that there must be sufficient information to create a symbolic rendition of each work, including rendition of anchors or link endpoints. So, if the structure container does not include works, then the links, anchors, collections, and revision selection rules held by the structure container must possess enough information to connect links to the works. If works are part of the structure container, then among the works, links, anchors, and revision selection rules, there must be enough information to connect a specific link revision to a specific work and/or anchor revision.

The *link traversal criterion* asserts there must be sufficient information to perform a link traversal from an anchor. If anchors are not part of the data model, then there must be sufficient information to traverse a link from the symbolic depiction of a link endpoint (e.g., some symbol that represents the endpoint of a link that connects entire works).

The primary elements of the structure design space are:

What does the structure container hold?

While the structure container must, at minimum, contain links, it is by no means limited to them. The structure container may also hold works, anchors, and other container objects. If the structure container only contains links, it is capable only of representing a link structure. If the structure container also holds works and anchors, it can represent not only the link structure, but also a consistent slice through a hypertext, holding not just the links, but also the linked works, along with the anchor points within those works.

If the goal is just to version structure, then the structure container need only contain sufficient information to satisfy both completeness criteria. So, if a link endpoint specifies an anchor revision, and an anchor revision specifies a work revision, then the structure container need only contain links, since it is possible, given a link revision, to determine the information needed to create a symbolic rendition, and to perform a link traversal. If, however, a link endpoint only specifies a versioned object, either the structure container or the containment relationship between the link and its containees (works or anchors) must hold a revision selection rule (discussed further below) to satisfy the completeness criteria.

Since a link in conjunction with a revision selection rule contains sufficient information to satisfy the completeness criteria, these criteria alone do not provide any motivation for adding works, anchors, or other container objects (e.g., collections, composites) into the structure container. However, most composite-based hypertext versioning systems do include works and links within composites, which act as structure containers. Here the motivation is to make the composite do dual duty, as both the structure container and as a workspace (discussed in Section 8.7.2). Workspaces provide the benefit of maintaining an internally consistent subset of the entire object space.

This point in the structure versioning design space is fully specified by giving a complete list of the entities contained by the structure container.

Versioning design space choice for structure container and its containees.

For the structure container, and all of its containees, one of the choices of the versioning design space—versioned object, within-object versioning, predecessor and successor relationships—must be made (Section 8.2.1 provides a complete description of the versioning design space, and Section 8.5.1 describes the link versioning design space). While it is mandatory for the structure container to be versioned in order to provide structure versioning, versioning for containees is optional. For example, both HyperPro [141] and HyperProp [178,179] do not version links individually, and version structure by placing the links inside containers that are versioned, and therefore each revision of the container records a specific revision of the link structure.

Though the versioning design space described in Section 8.2.1 includes an option for change-based versioning, to date no hypertext versioning systems have explored the use of change-orientation for structure versioning. If change-orientation we employed, it seems likely that the structure container and its versioned containees would all need to employ the same change-oriented versioning technique. Certainly the use of change-orientation for structure versioning remains an avenue for future research.

This point in the structure versioning design space is fully specified by listing, for the structure container and each of its contained entities, the choice of versioning mechanism employed to record the revision history of the entity. If the entity is not versioned at all, that is noted instead.

Containment design space choice for all container/containee pairs.

For each container/containee pair involved in structure versioning, their containment relationship needs to be specified by choosing a point in the containment design space, described in Section 4.2. This applies not just to every object contained by the structure container, but also to the endpoints of links (since links are modeled as containers), and to the containment relationship between anchors and their work objects. Furthermore, for each container/containee pair, if the containee is versioned using either the versioned object or within-object approach, it is necessary to determine whether the versioned object, or an individual revision, is contained.

Of the many choices inherent in each containment relationship, whether the containment type is inclusion or referential has the greatest impact on structure versioning, since inclusion containment implies that versioning of the contained item is dependent on versioning of the container. Referential containment leaves the containee free to have a revision history that is independent of the revision history of its container.

If the structure container only allows single containment of its objects, this leads to significant object duplication during the evolution of the structure, since every revision of the structure container constitutes a separate container, and singly contained objects can only belong to one. As a result, new structure container revisions that employ single containment must replicate all contained objects when a new revision, or working copy is made.

When the structure container holds the link and a single revision of its endpoint objects (anchor, work, or both), and the revision selection rule is located on the structure collection, an additional dynamic containment choice becomes available. Typically, a link referentially contains its endpoints, selecting a specific revision of endpoint objects using the revision selection rule. The structure container also evaluates the revision selection rule to select an individual revision of each endpoint object. Thus, the revision selection step is duplicated by the link and the structure collection. To avoid this duplication, the link could employ *indirect referential containment*, where the link endpoint is the revision selected by the structure container. That is, the link endpoint is a binding point that is filled-in by the revision selection rule evaluation of the structure collection. Figure 23 below shows an example of indirect referential containment. In Figure 23a, a containment diagram shows the containment relationships among objects. Each structure collection revision contains a work revision by evaluating a revision selection rule across a work's versioned object. The link uses indirect referential containment to select the same revision. Figure 23b shows an example instance of this containment structure, where link *l* is contained by structure container C. The link's endpoints are the revisions selected by the revision selection rule on C.

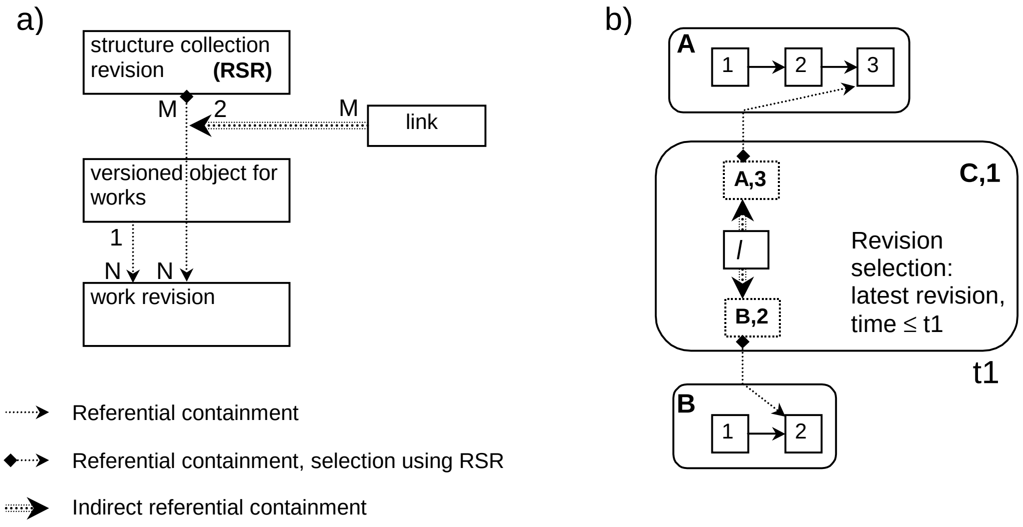

a)
structure collection revision (RSR)
M
2
M
link
versioned object for works
1
N
N
work revision
Referential containment
Referential containment, selection using RSR
Indirect referential containment
b)
A
1
2
3
C,1
A,3
l
B,2
Revision selection: latest revision, time ≤ t1
t1
B
1
2

Figure 23 – Indirect referential containment.

Location and scope of revision selection rule.

Dynamic containment using a revision selection rule (defined in Section 4.6) is often used in structure versioning, providing several benefits. Revision selection rules may be located either in the structure container, or on a specific containment relationship. When located on the structure container, the scope of the rule is all containment relationships where the containee is a versioned object (i.e., the container used in either the versioned object or within-object versioning approaches). When located on a single containment relationship, its scope is that relationship. The advantage of having the revision selection rule on the structure container is evaluation efficiency, and ease of maintenance. The advantage of having revision selection rules on each containment arc is flexibility, with each containment arc permitting a separate revision selection rule.

Revision selection rules bring several benefits. They allow more expressive selection of revisions than just explicit selection by revision identifier, permitting selection such as "most recent revision," or "most recent as of a specific time," or, in combination with a human-readable label (such as described in Section 8.2.6), "the revision with label *Beta_Release_2*".

Revision selection rules also allow a single unversioned link to refer to different revisions over time. This trick is accomplished by having the link endpoint be a versioned object, and then using the revision selection rule to select a specific revision. Since the rule is stored separate from the link, the rule, and hence the selected revision, can change without modifying the link. If the holder of the revision selection rule is versioned, the link then has the appearance of being versioned, since the selected revisions change over time with the revision selection rule.

Figure 24 highlights the link proliferation that can occur when links are unversioned, and revision selection rules are not used. A structure container, C, referentially contains two versioned objects, A and B. In this example, it is desirable to keep the link always pointing to the most recent revision of an object. In addition, old states of the link need to be preserved. Since the links are not versioned, and may only point to a specific object revision (i.e., they do not use revision selection rules to choose the revision they link), the only way to satisfy the criteria is by creating a new link. Thus, in revision 1 of C, link *l* is between the latest revisions of A and B, while in revision 2 of C, link *m* is added to link the new latest revisions of A and B, with *l* remaining unchanged to preserve the previous link endpoints.

In contrast, Figure 25 shows the use of revision selection rules with an unversioned link. In the figure, the structure container, C, referentially contains two versioned objects, A and B. By using the versioned object as the link endpoint, and then letting the revision selection rule choose the specific revision, the need to duplicate link *l* to point to newly created revisions is eliminated. Including time in the revision selection rules allows prior link endpoints to be recovered when reverting to a prior container revision.

Finally, revision selection rules on the structure container provide a single modification point for changing its contained revisions, a useful trait when performing time-based revision selection or label-based revision selection. Label-based revision selection has the additional benefit of creating internally consistent hypertext structures, assuming the hypertext was consistent when the labels were applied.

This point in the structure versioning design space is complete when, for each container/containee pair, including the structure container and its containees, as well as links as their contained endpoints, where the ultimate containee is a revision, a decision is made whether a revision selection rule on the structure collection, on the particular containment relationship, determines the revision endpoint of the containment arc. Alternately, if no revision selection rule is employed, meaning a specific revision is explicitly selected, this is noted as well.

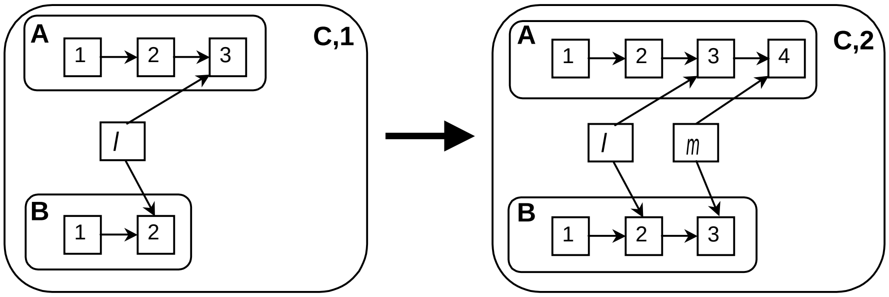


Figure 24 -- Link proliferation in the absence of revision selection rules.

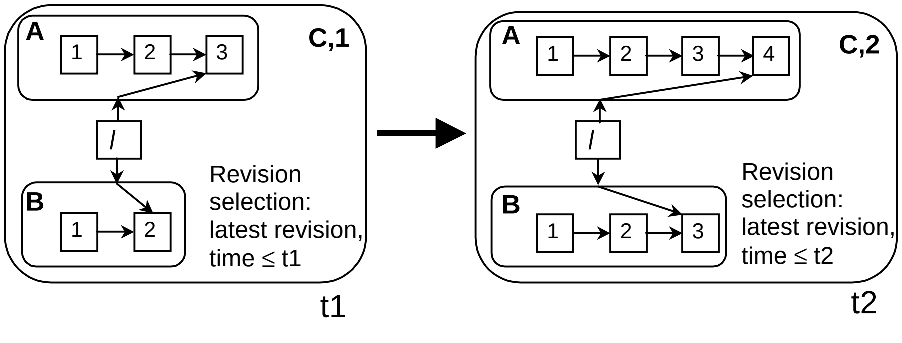

A
1
2
3
C,1
l
B
1
2
Revision selection: latest revision, time ≤ t1
t1
A
1
2
3
4
C,2
l
B
1
2
3
Revision selection: latest revision, time ≤ t2
t2

Figure 25 – A revision selection rule on the container chooses the revision for each link endpoint; link endpoints are versioned objects.

The following sections provide examples that highlight use of the structure versioning design space.

8.5.2.1 Versioned Structure with Unversioned Links

In this example, the following choices were made within the structure versioning design space:

- Abstractions present: structure collection, works, links. No anchors are present, links join whole works.
- Structure container contains: links ($D_l(t)$ or $P_{c,l}(t)$), work versioned objects ($V\text{-}O_w$)
- Versioning design space choices:
  - Structure containers are versioned, using versioned object approach
  - Links are *unversioned*
  - Works are versioned, using versioned object approach
- Containment design choices:
  - Structure container → link, work versioned object: *referential*, multiple containment, single membership, unordered, containment relationship on structure container
  - Link → work versioned object: *referential*, multiple containment, single membership, *ordered*, containment relationship on link (container)
- Revision selection rule: stored on collection, affects all link endpoints, provides selection of specific work revision from work versioned object (V_n from $V\text{-}O_w$)

A containment diagram showing these design choices is given below in Figure 26. An instance of this containment structure is shown in Figure 25 above. As discussed in the previous selection, this approach achieves structure versioning using unversioned links. Essentially, this approach stores separate revisions of links without explicitly recording their predecessor and successor relationships. Instead, the predecessor and successor relationships for the links are implicitly recorded by the structure container's revision history, since the unversioned links are contained by the structure container. As a result, each revision of the structure container records a specific revision of the link structure. This approach has the drawback that it is not possible to efficiently evaluate revision selection rules across the revisions of a specific link. This approach is used by both HyperPro [141] and HyperProp [178,179], because they focus on recording the history of link structure, not individual links.

The symbolic rendition completion criteria is met, since each revision of the structure container holds a versioned object for each work, a revision selection rule that selects the revision to display, and all links. Thus, for each work, all information is present that is needed to create a symbolic rendition. The link traversal criteria is also met, since the link endpoints, work versioned objects, are also present in each revision of the structure collection, and the revision selection rule chooses the specific work revision for each endpoint.

Variants of this unversioned link approach are possible, but have not been explored in the existing literature. For works, it is possible that within-object versioning could be used, since the structure container holds the versioned object, and its permits efficient evaluation of revision selection rules. Use of the predecessor and successor relationships approach for versioning works is not compatible, since links require efficient evaluation of revision selection rules.

There is really only one place where the revision selection rule can be stored, and that is the structure container. The link cannot store the rule, since that would entail creating a new link for every revision of the structure container in order to preserve its selected revision at the time the container was frozen. If the work versioned objects store the rule, it is impossible to freeze the rule when a new structure container revision is made, since a new revision of the structure container does not imply a new revision of its contained versioned objects, since they are unversioned.

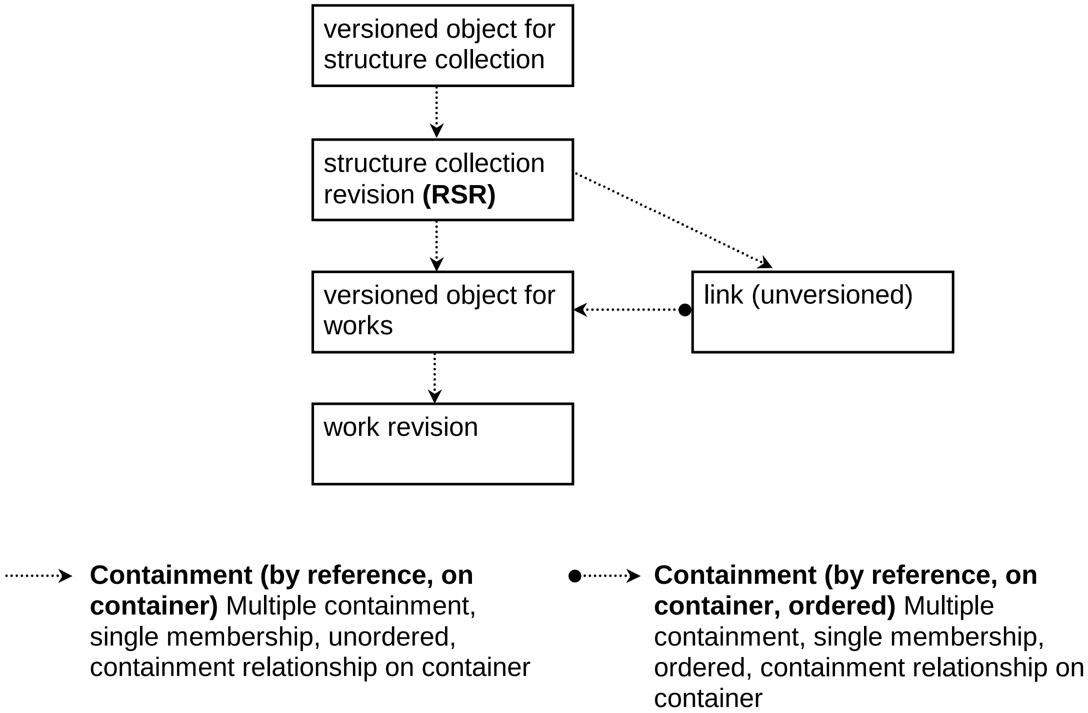

versioned object for structure collection
structure collection revision (RSR)
versioned object for works
link (unversioned)
work revision
Containment (by reference, on container) Multiple containment, single membership, unordered, containment relationship on container
Containment (by reference, on container, ordered) Multiple containment, single membership, ordered, containment relationship on container

Figure 26 – Containment diagram showing versioned structure using unversioned links, and a revision selection rule on each structure collection revision.

8.5.2.2 Versioned Structure with Unversioned Links Employing Indirect Referential Containment

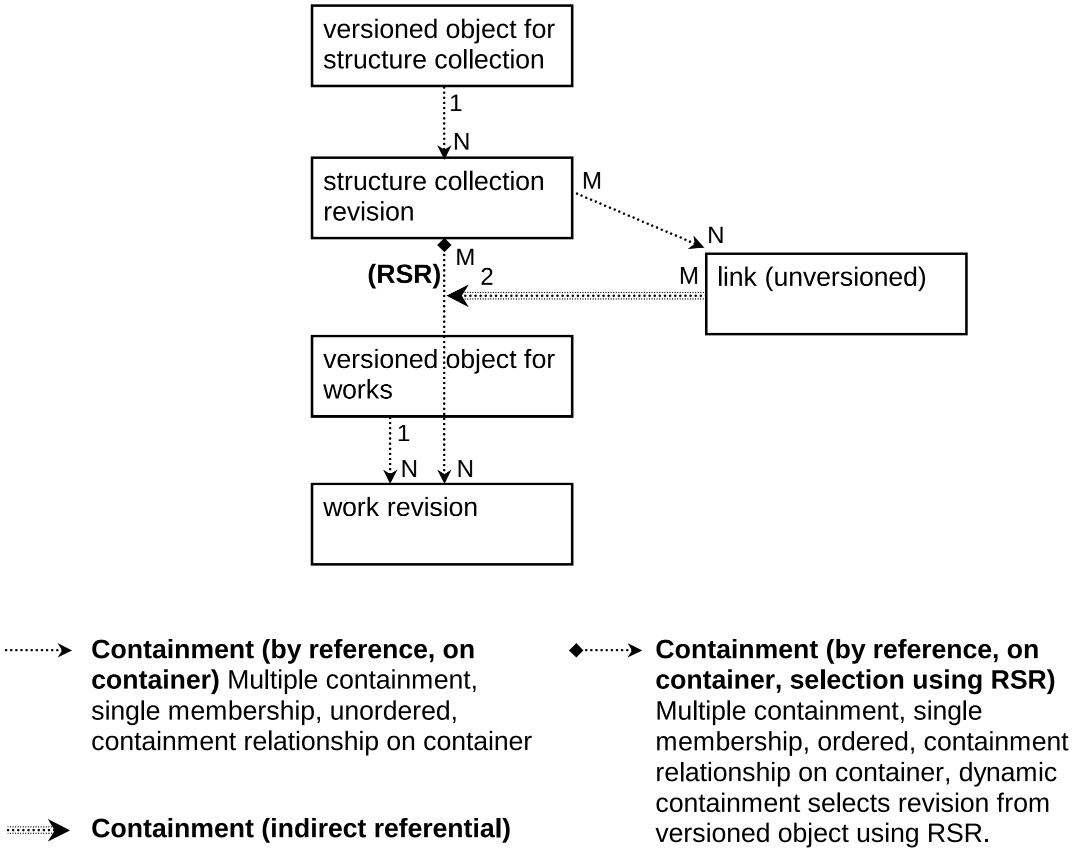
In this example, the following choices were made within the structure versioning design space:

- Abstractions present: structure collection, works, links. No anchors are present, links join whole works.
- Structure container contains: links (*$D_l(t)$* or *$P_{c,l}(t)$*), work revisions (*V_n* by revision selection on *$V\text{-}O_w$*)
- Versioning design space choices:
    - Structure containers are versioned, using versioned object approach
    - Links are *unversioned*
    - Works are versioned, using versioned object approach
- Containment design choices:
    - Structure container → link: *referential*, multiple containment, single membership, unordered, containment relationship on structure container

    - Structure container → work revision: *dynamic, referential*, multiple containment, single membership, unordered, containment relationship on structure container, dynamic containment via revision selection rule over work versioned object.
    - Link → work revision: *indirect referential containment*, selects containee of a structure container → work revision containment arc
- Revision selection rule: stored on structure container → work revision containment relationship, provides selection of specific work revision from work versioned object (V_n from $V\text{-}O_w$)

Figure 27 shows a containment diagram for these structure versioning design choices. In contrast to the previous example, here the structure collection contains work revisions, instead of the work versioned object. The link's containment is also different, employing indirect referential containment instead of containing the work versioned object. While this configuration has never been implemented in a hypertext versioning system, it would be expected to have the benefit of flexible revision selection, since each containment arc between structure container and work revision can have a separate selection rule. The inefficiency of evaluating a selection rule for each containment arc is mitigated somewhat by the link's reuse of the selection for its endpoints.

Figure 28 shows an example of this structure versioning approach. A structure container, C, referentially contains one revision from work versioned objects A and B. Revision selection rules on the individual containment relationships choose which revision is selected. The endpoints of link *l* are the same revisions, due to the link's use of indirect referential containment. Since the revision selection rules are part of the containment relationship, which is part of the state of the container, they are versioned along with the container. Hence the revisions selected by the link can change over time, without leading to link proliferation.



versioned object for structure collection
1
N
structure collection revision
M
N
M
(RSR)
2
M
link (unversioned)
versioned object for works
1
N
N
work revision
Containment (by reference, on container) Multiple containment, single membership, unordered, containment relationship on container
Containment (indirect referential)
Containment (by reference, on container, selection using RSR) Multiple containment, single membership, ordered, containment relationship on container, dynamic containment selects revision from versioned object using RSR.

Figure 27 – Containment diagram showing versioned structure using unversioned links that employ indirect referential containment for their endpoints.

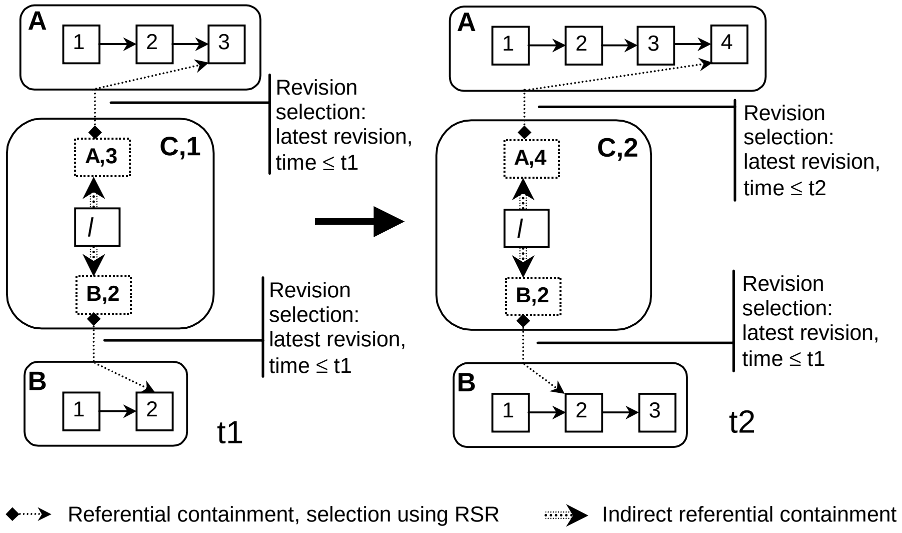

A
1
2
3
Revision selection: latest revision, time ≤ t1
A,3
C,1
l
B,2
Revision selection: latest revision, time ≤ t1
B
1
2
t1
A
1
2
3
4
Revision selection: latest revision, time ≤ t2
A,4
C,2
l
B,2
Revision selection: latest revision, time ≤ t1
B
1
2
3
t2
Referential containment, selection using RSR
Indirect referential containment

Figure 28 – Two revisions of structure container C, each containing an unversioned link, *l*, and two contained work revisions, one each from versioned object A and B, selected by the revision selection rule on the containment arc. The endpoints of the link are the same revisions, since the link uses indirect referential containment.

8.5.2.3 Versioned Structure with Versioned Links

In this example, the following choices were made within the structure versioning design space:

- Abstractions present: structure collection, works, links. No anchors are present, links join whole works.
- Structure container contains: link revisions ($V_{l,n}$), work revisions (V_n)
- Versioning design space choices:
  - Structure containers, links, and works are versioned, using versioned object approach
- Containment design choices:
  - Structure container → link revision, work revision: *dynamic, referential*, multiple containment, single membership, unordered, containment relationship on structure container, dynamic containment via revision selection rule over link and work versioned objects.
  - Link → work revision: *dynamic, referential*, multiple containment, single membership, *ordered*, containment relationship on link (container), dynamic containment via revision selection rule over work versioned object.

- Revision selection rule: stored on containment arc between structure container and its containees, providing selection of link revisions from link versioned objects ($V_{l,n}$ from V-O_l) and selection of specific work revisions from work versioned objects (V_n from V-O_w).

Figure 29 shows the containment diagram for these structure versioning design choices. The distinguishing element of this example is its use of versioned links, and the use of revision selection rules on all containment arcs of the structure collection revisions, and link revisions. An example instance of this structure versioning approach is shown in Figure 30. In essence, this is the structure versioning approach used by CoVer [87,88], and VerSE [91].

While the placement of revision selection rules on containment arcs yields excellent revision selection flexibility, it also has two drawbacks. First, evaluation of individual revision selection rules is less efficient than evaluation of one rule for all containment arcs. Second, it increases the work that must be performed to ensure the structure container holds a consistent hypertext. In Figure 30, consider if RSR3 were changed to be “latest, time ≤ t_2” and hence *l* ,2 selected A,3 instead of A,2. A link traversal across *l* ,2 starting from its other endpoint, B,2, would result in the display of A,3 even though the structure container current holds A,2. A user would perceive this as inconsistent.

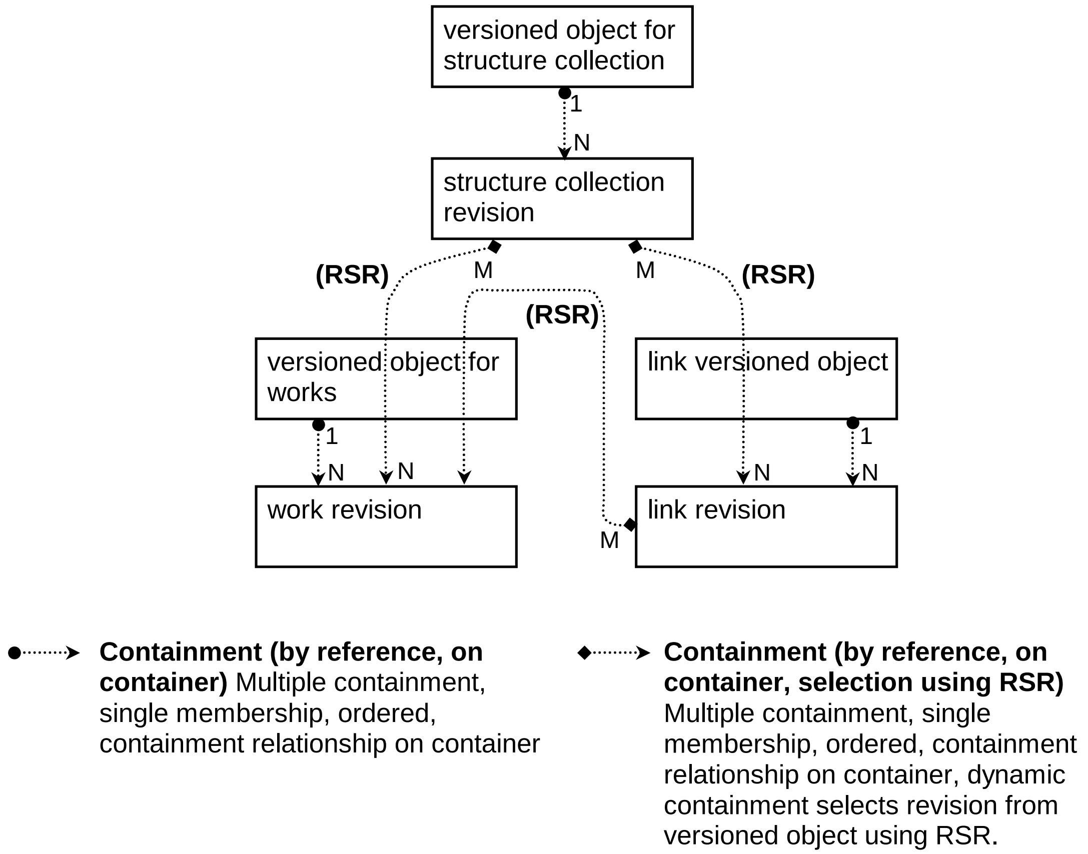

versioned object for structure collection
1
N
structure collection revision
(RSR)
M
M
(RSR)
(RSR)
versioned object for works
link versioned object
1
1
N
N
N
N
work revision
link revision
M
Containment (by reference, on container) Multiple containment, single membership, ordered, containment relationship on container
Containment (by reference, on container, selection using RSR) Multiple containment, single membership, ordered, containment relationship on container, dynamic containment selects revision from versioned object using RSR.

Figure 29 – Containment diagram showing versioned structure using versioned links. Structure containers and links use containment where the revision selection rule is on the containment arc.

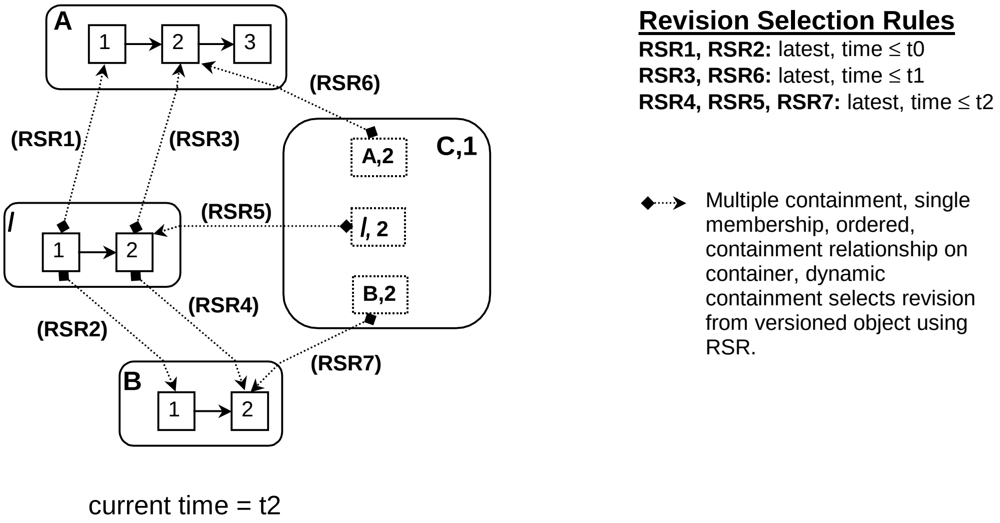

A
1
2
3
(RSR6)
(RSR1)
(RSR3)
C,1
A,2
(RSR5)
l
1
2
l, 2
(RSR4)
(RSR2)
B,2
(RSR7)
B
1
2
current time = t2
Revision Selection Rules
RSR1, RSR2: latest, time ≤ t0
RSR3, RSR6: latest, time ≤ t1
RSR4, RSR5, RSR7: latest, time ≤ t2
Multiple containment, single membership, ordered, containment relationship on container, dynamic containment selects revision from versioned object using RSR.

Figure 30 – An example of structure versioning using versioned links. The structure container, C, contains one revision from versioned objects A and B, and one link revision from versioned link *l*. Each containment arc has its own revision selection rule.

8.5.3 Linking to a Specific Revision

To allow linking to a specific revision of an object, somehow the link endpoint must identify an explicit revision. This can be accomplished by ensuring the object identifiers within a link endpoint are capable of identifying a specific revision. Alternately, the containment relationship between a link and its endpoint objects can be governed by a revision selection rule, as was shown in the previous section.

When the endpoint identifiers explicitly identify a revision, there are two ways to accomplish this goal. In the first, the identifier itself has the capability of identifying a specific revision. For example, object references in CoVer [88] are (*object identifier, version identifier*) pairs. Neptune [46] is similar, dividing its identifiers into (*global identifier, instance identifier*) pairs, with the instance identifier providing the ability to reference a specific revision.

However, in some cases a legacy identifier space exists that is versioning unaware, and cannot be changed to add version identifiers. Such is the case with URLs on the World Wide Web. In this case, to ensure a link can be made to every revision, each one needs to be given one of the versioning unaware identifiers. This is the solution adopted by the DeltaV protocol, where “repository” URLs are created for each revision. While the exact details will vary across server implementations, the repository URLs will either be found in a separate part of the URL namespace from where the authorable resources are located, or will be additional parameters added to the authorable URL. For example, if a Web resource is available at URL http://www.server.org/documents/report.html, then its revisions may either be given URLs such as http://www.server.org/repository/obj5432.r1 and http://www.server.org/repository/obj5432.r2 representing that the revisions are in a repository controlled part of the namespace, or they can have URLs such as http://www.server.org/documents/report.html;rev=1 and http://www.server.org/documents/report.html;rev=2 where the revision identification information is added as a parameter at the end of the URL. Some existing Web versioning schemes use the parameter add-on method [46,173,150], and, since they do not need to handle versioning of collections, they avoid this scheme’s drawback: identifying the revision of parent collection versions. This is difficult because the parameter at the end of the URL only modifies the leaf nodes, not parent collections.

8.6 Variant Support

As discussed in Section 2.3.2, variants express differences among a class of objects that are similar with respect to a given abstraction. Due to this difference in the face of similarity, it often occurs that variants share content. For any data object, there are two important classes of variants, those that share content, and those that do not. In software development, it is common for variants of a module to share significant portions of code, while natural language translations of a document typically share negligible content. For variants that share content, the major challenge for representing variants is how to express the separateness of each variant, while holding onto the common content. Stated more concisely by Mahler, the goal is, "keeping things together and telling them apart" [123].

The *multiple maintenance problem* highlights the importance of tracking commonality across variants [123]. When each variant is kept in an individual object, it is easy to tell the variants apart. But, if a subsequent change needs to be applied to multiple variants, the same change needs to be applied to multiple objects. This is tedious and, when performed manually, error-prone work. However, if the common parts of the variants are tracked, then making the same change across all instances involves only one modification.

Techniques for representing variants can be divided into those that represent variation for individual objects and for compound objects, discussed in turn below.

8.6.1 Individual Object Variation

There are two approaches for representing individual object variants, *variant segregation* and *within-object variants* [123]. In variant segregation, a separate instance of the object is created for each variant. This approach requires no knowledge of the internal structure of the variant objects, and hence a single mechanism can apply to any kind of content. Each variant is also completely isolated from other variants, and a change to one variant does not affect others. But, this same advantage is also a drawback, since variant segregation is susceptible to the multiple maintenance problem. In fact the multiple maintenance problem is an inevitable consequence of treating the internal object structure as a black box, since it implies that meaningful change tracking, and hence within-object commonality tracking, is impossible. Another drawback is the combinatorial expansion of variants with each additional axis of variability, with its concomitant increase in storage use. Due to its drawbacks, variant segregation is best used either when variants have negligible common content, or when the internal object structure is unknown.

One way to represent segregated variants is to represent an abstract object with a collection of variants. The revisions of each variant are then represented using either a versioned object, or within-object versioning, techniques described in Section 8.2.1. This provides good isolation of variants, and flexibility in how the revisions are represented, but has the drawback of requiring more objects, and more storage space. The CoVer system [87,88] places all object revisions and variants within a mob, a multi-state object. Each revision or variant of the object has a series of attribute-value pairs defined on it. Users in CoVer work within container objects called tasks, and objects are contained using referential containment where each containment relationship has an associated query that selects objects from the mob based on their attribute values. When used for revision selection, this query typically returns just a single object, but when arbitrary attribute queries are made, multiple objects can be returned, and these variant objects are all included in the task.

Another approach is to use links with a type of "is-variant-of" to reflect that one object is a variant of another. This can be viewed as using links instead of a container object to represent the collection of variants. Like the container approach, using links has good isolation and revision representation flexibility, but requires many objects and has poor storage. A commonly used scheme is to represent each variant as a branch of a revision history graph. Storage is very compact, since commonality across revisions and variants can be exploited in delta storage compression. However, since parallel development is also typically represented using branches of a revision history graph, it can be difficult to distinguish between a permanent variant and a pre-merge parallel development branch. Of course, all techniques that store each variant in a separate object are susceptible to the multiple maintenance problem.

Within-object variation represents all object variants within a single source object. Within the object, content chunks are tagged as belonging to a specific variant, with common content having no variant-specific tags. This is known as the *fragments-and-attributes* organization [123]. Using the tag values, it is possible for a program to extract a specific variant from the multi-variant object. This scheme avoids all of the drawbacks of variant segregation. The multiple maintenance problem is avoided, since common content can be edited in one place, and only one compact object is needed to represent the variants, since the combinatorial expansion is represented internally within the object. However, there are some drawbacks to within-object variants. One common example of within-object variation is C language preprocessor instructions. However, once there are more than a few C preprocessor instructions in a source code file, editing the source code with a preprocessor-unaware text editor can be confusing due to the obfuscation introduced by the preprocessor instructions. As the amount of variance increases, the source file can become incomprehensible, and unmaintainable. Another drawback is the difficulty of tracking changes to a specific variant. When the variant object has its revisions tracked using either the versioned objects or predecessor/successor relationships only approaches (described in Section 8.2.1) a change to a single variant is recorded as a change to the entire object, thus losing which specific variant was altered [123]. Finally, within-object variants requires not just knowledge, but design control over the internal structure of the contents, and hence would be challenging to use with preexisting content formats.

In the fragments-and-attributes organization of within-object variants, the use of arbitrary attributes provides the greatest flexibility for identifying the variant to which a content chunk belongs, since each axis of variability can be associated with an individual attribute. The P-Edit [111] and MVPE [166] systems support setting and retrieving of arbitrary attributes, and the C preprocessor can be viewed as supporting arbitrary attributes as well, since the number of possible symbol values in unconstrained. Delta [37] provides a more limited capability of setting a version string that associates a description of a volume within variant space with a content chunk. This approach is susceptible to long string lengths as the number of axes of variability increases, and it also depends on standard string values for representing points in variant space.

Revisions can be viewed as variants along the time and branch axes of variability. Thus, it is not surprising to see symmetry between variant segregation and representing a revision as a separate object, and between within-object variants and within-object versions. Variant segregation represents a variant as an individual object, just as the versioned object and predecessor/successor relationships only approaches do for revisions. Both with-object variants and versions exploit control over the internal structure of the object to represent either multiple variants, or multiple revisions within one object. Broadly, within-object variant and versioning share the same approach, dividing the content into attributed sections that are reconstituted using some extraction process. Within-object versioning systems tend to limit the settable attributes to just those necessary for recording revisions, while within-object variant systems leave the possible range of attributes unconstrained. Thus, within-object versioning systems can be viewed as a subset of within-object variant systems with attributes limited to those necessary for recording revisions, such as person making the change, time of change, and revision identifier.

8.6.2 Compound Object Variation

Compound objects, those that contain other objects using some form of referential containment, provide additional challenges to manage their variation. In software development, one compound object of interest is an entire software system, or subsystem, while for document management a book is another example. Compound object variation encompasses not only collecting together individual object variants, but also the changing structure of the compound object from variant to variant. The objects and links contained within a compound object can change across variants.

Within the hypertext versioning literature, compound object variation has not been addressed. While the CoVer [87,88] and Hypermedia Version Control Framework [100] have sufficient expressive power to model compound object variation, the literature on these systems does not describe any effort to do so, coming closest in the parallel development of tasks in CoVer. As a result, compound object variation for hypertext networks remains an open area of research.

Software configuration management systems have explored this area more thoroughly, using the following approaches.

- **Containers.** Individual variants for each object in a compound object are selected, and contained referentially within a container. This container then represents a distinct variant of the entire compound object. This approach was used by the Gandalf project [132], where a collection gathered all of the variants of a specific software project. This collection, known as a variant, was then subject to version control. This approach is described as “inverted” since variant selection is performed first, before revision control takes place. The composite-based hypertext versioning systems CoVer and the Hypermedia Version Control Framework, since they support complex queries on containment relationships, could potentially use this approach to handle compound object variation without any further modification to the system.

- **Change-based.** Change-oriented systems like PIE [80] and EPOS [119] provide good support for compound object variation. These systems employ first-class logical change objects that span multiple objects. In the case of PIE, these change objects are called layers, which are gathered together into contexts. Typically, layers are used to represent changes to the compound object, while a context represents an entire variant, although the distinction here is slippery, and a layer can be used to represent a variant as well. In EPOS, changes are attributed with the space of variance they represent, and changes can be combined using predicates over these attributes.
- **System model.** In software development, where the compound objects represent software systems, a single object, called a system model, can be made that describes the composition of the entire system. In this case, different system variants can be represented using different system model objects [123].

All of these techniques could potentially be applied in the domain of hypertext versioning systems, and exploration of each in this context remains an area for future research.

8.7 Collaboration Support

Almost all hypertext versioning systems, as well as document management, and configuration management systems, are designed to support group collaborative work. Collaboration can be loosely grouped into *asynchronous* and *synchronous* collaboration. Asynchronous collaboration typically involves collaborators working at different times, though it may occur at the same time, group calendars being an example. In contrast, synchronous collaboration involves collaborators working closely together in a tightly coupled, same-time session, exemplified by chat, and WYSIWIS (what you see is what I see) editors. Edwards additionally distinguishes *autonomous collaboration*, in which collaborators initially work independently on a shared artifact, but then come together for a period of more intense, tightly coupled work to integrate work done by individual collaborators. This type of work frequently occurs in multi-author paper writing without computer collaborative authoring tool support [57]. SEPIA seems well suited to support autonomous collaboration, since it supports smooth transitions from loosely coupled to tightly coupled collaboration [93].

For hypertext versioning systems, there are issues of collaboration-in-the-small, that is collaboration issues affecting just a single object, and collaboration-in-the-large, collaboration issues that span multiple objects. In the sections below, concurrency control techniques are discussed for handling collaboration-in-the-small, with larger scale collaboration is addressed with sections on workspaces, and merging of hypertext networks.

At its most general, collaboration support goes beyond the data management issues discussed in this section, and encompasses many user interface concerns. Conveying awareness of the state and activities of a collaborative group is a significant help in giving individuals the information they need to determine their next action. Awareness ensures that actions are relevant to the group's activity, and can be evaluated with respect to group goals and progress [53]. But, since little user interface work has been reported for hypertext versioning systems (notable exceptions being CoVer [87] and VerSE [91]), no clear design recommendations can be made here. This is certainly a topic for future research.

8.7.1 Concurrency Control

Concurrency control is one major aspect of collaboration support. When multiple people are working on the same object or link at the same time, concurrency control mechanisms act to ensure that each contributor's changes are not inadvertently lost. Concurrency control techniques can be evaluated using the following qualities: availability, transparency, consistency, responsiveness, and genericity. *Availability* concerns the ability to access an object for editing, and concurrency control schemes that limit access reduce availability. *Transparency* is an indication of the user visibility of the concurrency control mechanism. If a user is unaware a concurrency control mechanism is operating, it is considered to be wholly transparent. *Consistency* concerns the degree to which all collaborators share the same view of the object, with identical views being the most consistent. *Responsiveness* is the degree to which concurrency control affects the interactivity of the system's user interface [52]. Lastly, *genericity* is an indication of whether the technique is specific to only one, or certain kinds of objects, or whether it can be applied across all object types equally.

Existing concurrency control techniques provide various tradeoffs among these characteristics. Whole object exclusive locking (e.g., as used in WebDAV [201]) is a technique common to version control, and exhibits extremes of many characteristics. Once an object is exclusively locked, only the owner of the lock can write to the object, thus providing no availability for other collaborators. Locking is relatively non-transparent, since collaborators need knowledge about who currently controls the lock. However, locking provides a high degree of consistency, since only one collaborator has write access at any one time. Locking typically occurs before the object is displayed for editing, and hence is very responsive, and locking is also very generic, since it requires no knowledge of the object's internals, and hence can be applied equally to all kinds of objects.

Most concurrency control techniques provide greater availability than exclusive locking, and can support multiple collaborators working on the same object at the same time. During collaboration, each person works on his individual copy of the object. Until one collaborator's changes have propagated to all others working on the same object, these changes create a variant of the object. Concurrency control thus can be viewed as a mechanism for managing variants, where the axis of variability is the collaborator, the person making the changes. Unlike the variants discussed in Section 8.6, there is an expectation that collaborator variants will eventually be reconciled to produce a view of the object that combines the modifications of one or more collaborators.

A differentiator among concurrency control techniques is the typical length of time between the editing action that creates a variant, and the reconciliation of that change with all other collaborators, a time period Dourish terms the *period of synchronization* [52]. Roughly, techniques can be divided into "short", "medium" and "long" synchronization periods. On the short side are operational transformation algorithms, such as dOPT (used in GROVE) [58], that emit event messages to all other active collaborators very soon (typically less than a second) after an individual operation is performed. In this case, each operation creates an *ephemeral variant* that exists for a very short time before all other collaborators have a similar view of the object. Other concurrency control techniques maintain *temporary variants* that last a short time, on the order of minutes to hours, though sometimes as long as days. Duplex [144] and Alliance [165] both decompose an object into sub-parts, and then replicate those parts to each collaborator. Changes to parts may stay in a local replica for some time before they are communicated to other collaborators. Finally, techniques such as using branches in a version graph to represent parallel work explicitly represent each collaborator's work in a distinct persistent object that exists in perpetuity, creating a *permanent variant*.

Adopting the period of synchronization as the primary means of organizing concurrency control mechanisms (per-collaborator variant mechanisms) yields the following taxonomy:

No variants. Whole-object exclusive locking ensures that only one person has access to the object, and hence prevents per-collaborator variants [203]. Feiler notes that locking is the typical concurrency control technique for version control systems employing the checkout/checkin model [65], examples including the locking in SCCS [163] and RCS [185]. For a linear version history, locking typically implies that only one person may work on the graph at a time, though if revisions are mutable, it is meaningful to permit per-revision locking.

Ephemeral variants. As discussed above, operational transformation algorithms, such as dOPT, adOPTed [160], and GOT [183], are a concurrency control technique that uses ephemeral variants. Operational transformation distributes state replicas to all collaborators, and then sends real-time update messages from a changed replica to all others, thus making this technique suitable for synchronous authoring. Operational transformation algorithms provide high availability, since all collaborators can work simultaneously. They are reasonably transparent, although collaborators working in the same section will notice the contents changing due to edits by other collaborators. The algorithms have also been designed for a high degree of consistency. Their drawback is a lack of genericity, as the update events are dependent on content type. Most operational transform algorithms have been designed for synchronous text editing, and need extensions to handle other content, such as spreadsheets [145].

Temporary variants. Sub-object replication algorithms produce temporary variants. This can be viewed as a form of within-object variation, but at a very coarse grain size. Whereas most within-object variant schemes record variability in text objects at either a character or line level, sub-object replication divides an object into a small number of chunks, with each chunk containing a significant percentage of the object's state. Once divided, the chunks are replicated to each user's work site, increasing responsiveness.

Two choices present themselves for managing access to chunks, exclusive access, or unmoderated access. Alliance is an example of exclusive access, with each chunk having a writeable master copy at one collaborator site, with only read-only chunk replicas at other sites [165]. When a collaborator wants to work on a chunk, they petition to make that chunk the master at their site, which is successful if no one else is currently working on it.

MESSIE is another system that creates temporary variants with exclusive access [167]. Documents in MESSIE are subdivided into chunks, each chunk stored in a separate file. A master copy of each chunk is located at a central point. All access to document chunks in MESSIE is performed via email, which the MESSIE system scans for commands to checkout and checkin document chunks. Document chunks are locked while they are checked out, though read-only copies can be retrieved. Locks automatically time out after 48 hours.

As another example, Duplex can provide both exclusive access and unmoderated access [144]. In unmoderated access, overwrite conflicts can potentially occur, however, the expectation is that these conflicts will be discussed using the bulletin board feature which allows messages to be associated with the object being edited. A journal is also maintained for each object, recording modification operations to it during its lifetime, and this can help collaborators reconcile their local updates with the central object. Duplex also provides the unusual capability of having the access type vary from chunk to chunk within a subdivided object.

Sub-object replication with exclusive access provides moderate availability. When collaboration tasks have been divided well, and authors do not need write access to other sections, availability is high. But, when collaborators need frequent access to all parts of the document, the fact that other collaborators may have a chunk locked reduces availability. Transparency is medium, since collaborators need to know when a chunk is in use. Consistency is high, since only one person may modify a chunk at a time. Responsiveness is good when network connections are up, but if a network connection goes down, users will be subjected to network timeouts, and a loss of availability. Genericity is fairly low, since the technique depends on knowledge of the internal structure of the object to perform the decomposition, and the object must be capable of decomposition. When sub-object replication is used with unmoderated access, availability is increased, since any section can be worked on at any time, but at the expense of consistency, which is reduced due to the possibility of overwrite conflicts.

Sub-object locking is another concurrency control technique that produces temporary variants, each variant held in the memory of the authoring tool until changes have been written to the partially locked master copy [203]. Its characteristics are similar to those of sub-object replication with exclusive access, providing moderate availability, medium transparency, high consistency, and low genericity. Sub-object locking can be viewed as being similar to sub-object replication with exclusive access, except that replication is handled by the individual applications when they copy the centrally stored object into memory, instead of the object store performing the replication service.

Permanent variants. Concurrency control mechanisms that employ permanent variants can be divided into those that employ variant segregation, placing each variant in a separate object, and those that employ within-object variation, storing all variants within the object itself.

Permanent variants using variant segregation. Just as branches of a version history are used to represent other kinds of variants, so too are they used to represent collaborator-specific variation. In the branching scheme, when multiple people are working on a versioned object, a checkout creates a new branch for each collaborator, and does not lock the version history. Each branch thus represents the changes made by a specific person, and the act of simultaneous work has traditionally been called *parallel development*. Since changes are stored as revisions, the variants are persistently recorded. Once collaborators have finished working in isolation, they combine their changes together using a merge tool to create a new revision. Many state-based version control systems employ this approach, Continuus being one example [36]. Composite-based hypertext versioning systems frequently use this technique for representing the changes made to an object that is held within a workspace container. The workspace contains multiple objects, and is associated with a specific collaborator. For each object a person has modified, the workspace contains a revision from the branch holding his current work. HyperPro [141], HyperProp [178], CoVer [89], and VerSE [91] all use this approach, and the facilities of the Hypermedia Version Control Framework [100] can be used to implement this scheme, as well as whole-object exclusive locking.

The concurrency control provided by CVS can be viewed as a type of branching scheme [20]. CVS stores the per-user variants within each user's working area, as opposed to the branching schemes above where the variants are stored within the version history. CVS can operate in either local mode, where the variants are located in a user-specific portion of a shared filesystem, or in remote mode, where the variants are retrieved from a central repository and stored on the user's local filesystem. Upon checkin, changes in the user's local workspace are merged with the current revision in the CVS repository.

Using branches to represent collaborator-specific variation provides high availability and consistency, with low transparency, since collaborators must explicitly check out an object, thereby gaining exclusive access to it. Responsiveness is high, since there are no other accesses that would diminish it. Genericity is also high, since no knowledge of internal object structure is needed to create a whole-object variant.

Permanent variants using within-object variation. Variant representation techniques that store all variants of an object within the object itself can represent the per-collaborator variants that emerge during simultaneous work. As described in Section 8.6.1, within-object variants employ a fragments-and-attributes organization, subdividing the object into fine-grain units, each of which has associated attributes that identify points along axes of variation. Often, these systems have a predefined axis of variation that is the person who made the change. VTML [194] and Palimpsest [56] are two data formats that can represent fine-grain changes and associate them with an individual person, although they are not very flexible for representing other types of variants.

The fragments-and-attributes technique provides excellent availability, since all collaborators can have access to the object at the same time, and their modifications are stored when they are made. Transparency is high during editing, since there is never a time when the object cannot be edited, but is low when merging together the work of several collaborators, since each collaborator's changes must be made visible during this operation. Consistency is high, since all changes by all collaborators are stored within the object, and hence are never lost. Responsiveness is high, since changes can efficiently be added to the object. Genericity is low, since the internal structure of the object must be modified to accommodate storage of within-object variants.

Period of synchronization	Mechanism	Advantages & Disadvantages	Example Systems
No variants	Whole object exclusive locking	Availability: none Transparency: low Consistency: high Responsiveness: high Genericity: high	SCCS [163], RCS [185], WebDAV [79], HyperWave [107], DistEdit [109]
Ephemeral	Operational Transformation	Availability: high Transparency: medium Consistency: high Responsiveness: high Genericity: low	dOPT/GROVE [58], adOPTed [160], GOT/REDUCE [183], shared sc [145]
Temporary	Sub-object replication (exclusive access to chunks)	Availability: moderate Transparency: medium Consistency: high Responsiveness: medium Genericity: low	Alliance [165], Duplex [144], MESSIE [167]
	Sub-object replication (unmoderated access to chunks)	Availability: high Transparency: medium Consistency: medium Responsiveness: medium Genericity: low	Duplex [144]
	Sub-object locking	Availability: high Transparency: medium Consistency: high Responsiveness: medium Genericity: low	Discussed in [203]
Permanent	*Variant Segregation* Branch of version history	Availability: high Transparency: low Consistency: high Responsiveness: high Genericity: high	HyperPro [141], HyperProp [178], CoVer [89], VerSE [91], Continuus [36], CVS [20], DeltaV [199]
	Within-object variation Fragments-and-attributes	Availability: high Transparency: high (editing), low (merging) Consistency: high Responsiveness: high Genericity: low	VTML [194], Palimpsest [56]

Table 8 – Concurrency control techniques organized by how long they maintain variants created by individual authors.

8.7.2 Workspaces

Workspaces provide three main benefits: a view of a shared hypertext network (or of a collection of objects) that is specific to a particular person, work isolation from other collaborators, and the ability to work in parallel on objects in the workspace. These are identified as *sharing, isolation,* and *collaboration* by Estublier [61].

In essence, a workspace is a container object that holds the objects comprising a hypertext network. One workspace is assigned to each collaborator, and frequently one or more additional workspaces act as a shared space where completed objects are kept. Examples of these per collaborator workspaces, and a shared workspace are both shown in the scenarios in Section 6.3. Object sharing is accomplished by using referential containment to multiply contain objects from the pool of objects managed by the repository, the approach adopted by CoVer [87], HyperPro [141], and HyperProp [179], or by making one copy of each object per workspace, the approach used by Neptune [46]. Isolation is achieved by ensuring that modifications to the shared objects are only visible to the person making the change, until he decides to share these changes with other collaborators. To ensure work isolation in the case of workspaces using multiple containment, work isolation requires that an object be replaced with a copy if it is to be modified. Work isolation is achieved automatically when sharing is achieved with object copies.

Composite-based hypertext versioning systems all use addresses to identify their objects, and hence avoid namespace issues when achieving work isolation. When a hierarchical namespace is used to identify objects, such as pathnames in a filesystem, work isolation additionally requires that each workspace be mapped to a different location in the namespace. Typically this is accomplished by re-rooting the namespace tree rooted at the workspace's top collection. For example, if the object tree originally begins at "/home/projectX/…" then one potential namespace mapping of the workspace for each collaborator would be "/home/{collaborator}/workspaces/projectX/…"

Achieving collaboration within workspaces involves application of the concurrency control design space, given in Section 8.7.1. While workspaces can be used with any concurrency control mechanism, they make the most sense when used with permanent variant schemes. Temporary and ephemeral variant mechanisms, such as operational transformations, and sub-document replication, do not require or encourage work isolation among collaborators. Instead, they make it possible to have multiple people work together on the same object. Thus, when concurrency control produces only temporary and ephemeral variants, all workspaces would tend to share the same objects. This eliminates the need for using workspaces to ensure work isolation. But, container/workspaces are also used to represent a sub-section of a larger hypertext structure, as well as a consistent time slice through that sub-part, thus highlighting the utility of containers even if they are not used for work isolation.

Using a branch of the version history to handle concurrency control is the mechanism employed by most composite-based hypertext versioning systems. Since these systems provide versioning services for a wide range of object types, this limits them to concurrency control with high genericity. Only whole-resource exclusive locking and version history branching satisfy this constraint. Since locking does not represent any per-collaborator variation, only branching satisfies the needs of composite-based hypertext versioning systems. To date, no composite-based hypertext versioning systems have used ephemeral or temporary variants (with the possible exception of CoVer [87], which, since it is based on SEPIA [93], might provide both loose and tight collaboration during cooperative editing). Despite the loss of genericity, it would be worthwhile to explore the use of temporary and ephemeral variants in composite-based hypertext versioning systems, since this would a better separation of concerns, separating work isolation from partitioning the hypertext and providing a consistent time-slice.

8.7.3 Merging Hypertext Networks

When containers are used in conjunction with a concurrency control scheme that employs permanent variants, individual collaborators work in isolation until they need to produce a single, coherent view of their work, typically for external consumption. In composite-based hypertext versioning systems, collaborators work in isolation on containers that hold hypertext networks until they need to produce a coherent merged hypertext network. The scenarios in Section 6.3 show an example of the merging of hypertext networks taking place between time t5 and t6.

Despite the importance of this activity, very little research has been performed in this area, making it difficult to give general design guidance. Exceptions to this rule are the preliminary work by Haake, Haake, and Hicks [90] that directly addresses the issue of merging hypertext networks, and the discussion by Delisle and Schwartz on the semantics of context merging in the Neptune system [46]. Merge support is required both for the contents of individual objects, and for the entire structure. Merge support for text objects is a staple feature of configuration management systems, and merge support for non-textual objects has also been explored, but not to the same depth, one example being Timewarp [57].

Project merge support in configuration management tools can be used as a starting point for investigation of hypertext network merges. The Project Revision Control System (PRCS) provides a merge tool that handles both within-file content differences as well as whole-file differences, such as a file added, deleted, or renamed in project revisions [121]. Adele provides similar merge support, centered on workspaces [61]. The merge algorithms employed by these tools can be used as a starting point for handling merging of objects and links. However, since links contain multiple objects, once objects have been merged, an additional consistency maintenance step must be performed to ensure that operations on individual objects and links do not cause dangling, or otherwise broken links.

In their exploratory work on hypertext network merging, Haake, Haake, and Hicks consider three different merge tools for the CoVer system, called the list-merger, the graph-unification-merger, and the graph comparison merger [90]. CoVer uses an all-attributes organization for data, and every object and composite contains a set of attributes that hold its state, including the "content" attribute. For composites, the content attribute holds a list of the objects, links, and other composites it contains. The columnated lists inherent to Smalltalk systems (other examples in the hypertext world include PIE [80] and Neptune) inspire the layout of the list-merger, which contains columns for selecting the object to be merged, the attribute to examine, one column for the attribute value in each of the predecessor revisions, and one last column for the final, merged value of the attribute. Above the columnar list is a graphical depiction of the version history for the object. Structure merging is possible using the list-merger by performing a merge on the content attribute of a composite. However, this does not provide a good visualization of the two structures being merged, or of the merged structure, and this motivates the next two merge tools.

The graph-unification-merger provides a graphical display of the union of two hypertext networks being merged. If multiple revisions for the same object occur due to the union operation, they are displayed in one of two ways. In the first visualization, alternate revisions are piled on top of each other, slightly offset, like playing cards. The second visualization uses a single screen object with tabs on its bottom, one tab for each alternate revision. Either by selecting one of the piled revisions, or one of the tabs, just one of the alternates is selected to participate in the final merged network.

The graph-comparison-merger also has two alternate visualizations. In the first, there are three windows, one each for the two revisions being merged, and the third showing the merged network. The second visualization uses overlays, with varying gray scales representing the initial versions and the merged version.

Unfortunately, these merge tools were not implemented, and no use experience was ever collected. Given the importance of the merge activity for supporting collaborative work in composite-based hypertext systems, it is important that future research be performed to flesh out the design parameters for hypertext network merging.

8.8 Navigation in the Versioned Space

Due to the emphasis of the hypertext versioning literature on data modeling issues, there has been very little research that focuses on the user experience while performing hypertext navigation in a space of versioned works, anchors, and links. As a consequence, there is a minimal experience base that can be used to provide design guidance for meeting the requirement for versioned hypertext navigation.

A starting point for consideration of this issue is recognition that there are two notions of time at play within a versioned hypertext, that of wall clock time, time that is read from a clock, and revision time, the sequence of revisions of an artifact. Consider a linear revision history of 3 revisions. When visually depicted, this revision history will typically use the same amount of screen space to show the revisions whether they were made over a span of 5 minutes, or 5 months. By contrast, a depiction that emphasized wall time would space the revisions proportional to elapsed time, and hence revisions 5 minutes apart would be shown much closer than revisions 5 months apart. This difference in depiction highlights that the increments of revision time are saved discrete states, not the minutes and hours of wall time.

Navigation in both times can be supported. The work by Feise on a prototype Web “way-back” machine focuses on wall time navigation [66]. The user of a way-back machine enters a date and time, such as “July 25, 1998, 11AM”, and then navigates through the Web as it was at that moment. Ideally, the user interface constantly displays the current navigation time, so someone returning to a browser after a time away will not confuse historical content as current. In contrast, the V-Web system provides a way to perform *revision time* navigation of versioned Web pages [180]. V-Web adds to the top of a Web page a frame containing a textual depiction of the page’s revision history, with links off to each revision.

Navigation in revision time is more typically supported. Composite-based hypertext versioning systems use one revision of a composite as the navigation context. Once this context has been set, all navigation uses the revisions of the works, anchors, and links within the context. While this provides a consistent navigation experience, it does not handle links whose destination is outside the composite. Furthermore, it seems reasonable that for each visible symbolic rendition, the user interface should display its point in revision time (i.e., its containing composite's revision). CoVer [87] and VerSE [91] are the only hypertext versioning systems that provide screen shots of their systems, and unfortunately they are unclear as to what additional navigation support they provide for versioned hypertexts. It is possible this support may not be necessary in practice. Software configuration management systems often make revisions available via standard file system interfaces, which, since they do not provide any revision identification information for either individual revisions or configuration revisions, make it impossible to provide revision time information in the user interfaces of tools, especially editors. Yet, despite the lack of this information, people are still able to get their work done. Still, the common technique of embedding a system-maintained revision identifier in a comment at the top of a text file suggests that even in software configuration management systems, awareness of the current revision time is necessary.

Though the lack of prior work in this area is a problem for implementers, it is an open field for further research.

8.9 Searching

The brute-force way to provide search capability across a versioned hypertext repository is to develop a search mechanism from scratch, using techniques from the database, or information retrieval community (this appears to be the approach used in Neptune/HAM [45]). However, since this is a time-consuming approach, systems builders often use a database as an infrastructure component on which they build hypertext object management capabilities. This is the approach adopted by CoVer [87], and the Hypermedia Version Control Framework [100], among others.

Web search engines highlight another approach, which is to delegate the search capabilities to an external repository. In this scheme, information about works are gathered and replicated in an external repository, which is tailored specifically for high-speed searches. This is a valuable approach when the works to be searched are spread across multiple distributed repositories that do not communicate with one another, and hence are incapable of supporting cross-repository searches. Furthermore, when works are distributed, the search must be distributed as well, incurring a performance penalty relative to a centralized search.

The Document Management Alliance (DMA) 1.0 specification provides another searching architecture [50]. In the DMA approach, a middleware layer coordinates a search across multiple document management repositories. DMA provides a mechanism for denoting equivalence of metadata items across servers, so a query can be processed even when the underlying metadata schemas differ.

8.10 Visualizing the Versioned Space

Containers within hypertext versioning systems are the main focus of visualizations. Today, most commercial configuration management systems visualize revision histories, and configurations, both container objects. However, there are no published surveys of this user interface work, and hence there is not much guidance that can be provided for their construction. There are instances of graphical revision history trees, with boxes representing revisions, and lines representing predecessor/successor relationships, and there are also examples of textual printouts of the same information. For graphical revision history trees, graph layout to achieve an aesthetically pleasing display is an engineering challenge. Koike and Chu present a 3-D visualization of revision histories and projects in [110], and provide evidence suggesting that their visualization allows faster initiation of checkin and checkout operations over the command line. To date, there are no published evaluations of revision history visualizations, and hence no data for basing a decision on which kind of interface is better for various tasks and user experience levels.

Hypertext versioning introduces the new concerns of how to visualize containers that hold both work and link revisions, such as composites, and how to visualize versioned links, and collections of links (structure containers). Much work has been performed on visualizing unversioned hypertexts; Durand and Kahn provide a taxonomy of unversioned hypertext visualization techniques divided into “graph-based structures, such as webs, hierarchies, and acyclic graphs; and spatial structures such as neighborhoods and abstract metrics” [55], p. 68. In the hypertext versioning literature, only CoVer [87] and VerSE [91] have screen shots of composite visualizations. CoVer and VerSE make use of a number of specific browsers, such as the “mob browser” (for revision histories), the composite browser, and the task browser. The revision history can be specialized to show the subset of the revision history that appears in a specific composite, using a “compound derivation” relationship to represent multiple revisions and predecessor/successor relationships (see Figure 2 in [87]). CoVer highlights the utility of multiple views over the same information space. In all of the CoVer and VerSE visualizations, works are represented as blocks, and links are lines between the blocks. While the blocks can grow and shrink depending on the number of objects on-screen, it is unclear how well their visualizations scale to large numbers of works and links.

Most visualizations of versioned information concentrate on depicting works and predecessor/successor relationships. However, other depictions can be quite useful. The WebGuide system provides a visualization of the changes in the neighborhood of a Web page using colors and shapes (oval vs. rectangle) to represent types of changes in Web pages, and dashes vs. solid lines to represent changed and unchanged links (Figure 6 of [51]). SeeSoft visualizes multiple versioned text files by condensing the text into a narrow column, and then color-coding the contents according to age, red for newest, blue for oldest, with a rainbow scale in-between [13].

Despite the scarcity of existing work on visualizing versioned hypertexts, any usable hypertext versioning system needs one or more ways of depicting the versioned space. Research on ways of visualizing versioned hypertexts, as well as on understanding the tradeoffs between visualization techniques, would be a significant help to future systems builders.

8.11 Traceability

When works are represented as compound documents, comprised of multiple objects, it is possible for a single object to be used in multiple composites. An object could be reused in multiple revisions of the same work, or it could appear in a completely different work. In essence, tracing the use of an object across multiple compound documents requires answering the question for a given object, “which compound documents contain me?” There are several ways to generate an answer.

If compound documents contain their objects referentially, and all compound documents are within the scope of a search facility, a conceptually simple solution to the traceability problem is to perform a search across all composites for those that contain a given object. Alternately, if it is possible to store a pointer back to a containing compound document every time an object is used, and remove the backpointer when the container is destroyed, then use tracing is just an enumeration of the backpointer list. Unfortunately, when objects are distributed across multiple trust domains, as is the case on the Web, many of the preconditions for these techniques no longer apply. Distributed objects are probably not all within the scope of a search facility, and it is likely that backpointers cannot reliably be stored, and backpointer maintenance cannot always be performed, due to the possibility of network outages, or lack of access rights.

Solutions to the distributed traceability problem will likely always be approximations. Searching for compound documents that use an object can still be performed using a technique similar to those employed by Web search-engines, where a large centralized search facility continually polls all reachable content to construct a model of the current state of the hypertext. However, this technique usually does not reach all readable content, and the central model is never current [113]. Still, a query of such a service could provide useful answers to the traceability question, even if they are not complete.

When compound documents are under version control, it raises the possibility that, once a single revision of a compound document is found to use an object, other revisions do too. The revision history of a compound document can be searched to find which revisions use a particular object. The revision history information can be used to supplement the other techniques described in this section.

Alternately, it might be possible to construct a compound document renderer that, every time it accesses an object, also transmits the identifier of the compound document. It would then be possible for the owner of the document to trace usage, though this would not help other parties.

8.12 User Interaction

The perceived complexity of a hypertext versioning system is dependent on its user interface, and its visualization of the versioned hypertext structure, and the operations used to manipulate it. To date, there has been little published research on the user interface of hypertext versioning systems, with Neptune [45], CoVer [87], and VerSE [91] providing the only examples of published articles containing screen shots. Since perceived user complexity is such an important issue for the adoption of hypertext versioning systems, additional research that focuses on just the user interface aspects would be very valuable.

User interface research could address an open question: is hypertext versioning technology simple enough that it could one day be used in mass market software systems, achieving ubiquity similar to that of word processors today? Or is it the case that hypertext versioning will only be of interest to a highly skilled user base that is willing to invest significant time in learning the concepts and operation of the technology, because their problems are too big, or the pain of not versioning is too high, or there is some compelling regulatory or business driver. At present, there is insufficient data to settle the question, since no system that supports versioning of structure and works has ever been used outside of a laboratory setting. Some parallels can be drawn with configuration management systems, which address similar problems, and are in wide use today. Configuration management systems provide work isolation, merging, problem tracking, and the ability to revert to prior revisions, as do the composite-based hypertext versioning systems, so in many respects the systems are of equal complexity. While configuration management systems do not explicitly version link structures, and thus are less complex, they do provide the ability to construct arbitrary configurations, a distinct increase in complexity over hypertext versioning systems that do not have this feature. This argues that there is a niche for hypertext versioning at least among highly skilled users.

But, what about less skilled use, such as in the home, or casual office use? The recurring pattern of document processing tools providing only linear versioning implies a tradeoff between data model complexity and breadth of users. Schedule demands can also apply pressure to simplify even skilled use of configuration management technology [196]. This implies a need for hypertext versioning systems to provide functionality layers, with a simple layer providing a minimal set of high value operations, and one or more layers giving more complete, and hence complex functionality. Such a layered strategy is used by the DeltaV protocol [199].

De-emphasizing user naming of items carries with it some additional tradeoffs. When the system is primarily responsible for naming, the entire issue of item naming tends to be minimized. Yet, as the Web has ably demonstrated, object identifiers, such as the URLs commonly found in all manner of advertising, are not internal to a system. Item names have an important role in creating namespaces that are shared by other users of the system. As hypertext systems become more distributed, naming issues become more important, and, likewise, the ability of the system to completely hide naming issues from the user is reduced. Instead of trying to completely remove user control over naming, a better goal is to assist with naming, suggesting names that authors can change at will.

Difficult naming issues are raised when versioning data or structure. The first issue is which items should have distinct names. Certainly each revision should be individually referenceable, but typically it is the versioned item which is named, and each revision has a name that combines the versioned item name with a revision identifier, as in “report,v5”. Containment adds only difficulty. When containers are named and versioned, one mechanism for naming their contents is to list the container’s name and revision, followed by the contained item, as in “folder,v2/report,v5”. Unfortunately, URLs cannot handle this. URLs are syntactically incapable of associating revision identifiers with interior path segments, thus highlighting the need to design versioning support into names from the beginning, instead of retrofitting them later.

With nested containment, names with explicitly listed revision identifiers quickly become cumbersome, and brittle in the face of change. More durable names will contain version selection criteria, as in “folder/report;(selection rule=‘latest as of Nov. 5, 1996’)”. This immediately implies that names are no longer unique, as multiple revision selection rules can identify the same item. Furthermore, it is a very slippery slope from simple revision selection rules to full query language support, and the attendant challenge of encoding that query into a human-readable name.

8.13 Tool Interaction

Introducing a separation of concerns between the repository and the user interface provides the benefit of freeing each half to focus on optimizing its capabilities. It also requires the creation of a programmatic interface between the user interface and repository layer, thus opening the possibility of multiple applications using the same repository. This is the essence of the open hypermedia approach, creating an open, standard interface to the hypertext functionality layer of a system. When the repository layer provides storage for just the anchors and links comprising the hypertext structure, it is termed a *link server* architecture. If general object storage is added, it becomes an *open hyperbase* [142]. Since hypertext versioning systems provide versioning services for objects, they have tended to adopt the open hyperbase approach; this is the case for CoVer [87], VerSE [91], Neptune [46], and the Hypermedia Version Control Framework [100]. This section primarily addresses the issues of tool interaction with an open hyperbase system, while the following section (§8.14) addresses the specific issues that arise in link server systems when object storage is separated from anchor and link storage.

Following the framework described in the NIST/ECMA reference model for integration in software engineering environments [64], there are five primary axes of integration:

Data integration: sharing information from multiple, heterogeneous repositories and sources.

Control integration: access to the facilities of an application without modifying its executable image, such that the capabilities of multiple applications can be flexibly combined.

Presentation integration: providing consistent appearance and interaction styles across applications.

Process integration: controlling applications based on an explicit description of a work process, and the ability to coordinate multiple processes across applications.

Framework integration: the degree to which applications make use of common facilities, such as authentication, security, etc. that exist within a specific integration framework.

The first three, data, control, and presentation integration, are the primary axes that impact application integration within hypertext versioning systems. A complete integration with a hypertext system is defined as a user interface for manipulating anchors and links, and an unbroken bi-directional path for communication between the application and the open hypermedia system [198].

Data integration is typically achieved by the open hyperbase creating an externally visible application program interface (API) usable by multiple applications. Creating multiple program language bindings to the interface increases the number of possible applications that can use the API, since it shifts the burden of creating a new language binding away from the tool integrator. When the API transmits commands and responses across a network, it permits the applications and repository to exist on separate machines, increasing scalability. If clients are limited in the external communication mechanisms they support, support for multiple network protocols can increase the pool of potentially integrable applications.

There are three primary architectures used in data and control integration [198,44]. A *custom* integration is one where the application's source code has been modified, or code has been written using the application's built-in customization language. This approach provides a large degree of control over the application, and hence the possibility of a very fine-grain integration. The *wrapper* approach places a mediator between the application and the hypertext system. The wrapper acts as a communications translator, converting between the interfaces and communications media supported by the hypertext system and the application. The *launch-only* approach integrates applications that are non-communicative, possessing neither a customization language, nor any external interface. Upon link traversal, the application is launched with the object representing the link endpoint.

Presentation integration is an area where application integration can run into difficulty. Applications typically exert significant control over their user interface, permitting some modification of key bindings and menu items, but no significant changes to the appearance or interaction styles. This is usually wise, since it enforces a degree of uniformity over the user experience, making it possible to provide manuals and telephone support. But, it runs counter to the desire of environment builders to create a uniform user interface and interaction style for cross-tool capabilities like creating links, and common versioning facilities such as check-out, check-in, etc. Application integrations must provide a visual depiction of link endpoints, yet application capabilities vary for accomplishing this task, leading to inconsistent depiction of link endpoints across applications. In an example of integrations with the Chimera system, a text editor is capable of underlining link endpoints, while a word processor cannot [198].

Versioning adds some additional burdens for application integration, since versioning requires the display of more information to orient the user in the time dimension. Ideally tools should provide answers to questions such as, what version is the user working on, what version will the user be traversing to, and what version will the user be creating a link to? The creator of a hypertext versioning system can provide user interface guidelines describing the information that should be provided, so applications can effectively orient the user. In order to minimize the integration burden for applications, this orientation information should be kept minimal.

Visualization of the versioned hypertext is another new user interface requirement for open hyperbase environments. It is beneficial to create specialized tools for visualizing revision histories and versioned composites and link structures, since this maintains a separation of concerns between visualization tools and other applications. It is also pragmatic: the integration facilities of most tools are insufficient for creating sophisticated visualizations. Content merge tools are a particularly thorny problem, since merging of content requires an understanding of the internal organization of the objects being merged. Usually the application has this knowledge, with the hyperbase treating the object as opaque data, and hence this would imply the application should provide a merge tool specialized for its objects. But, in practice, many applications do not provide their own merge capabilities. This suggests that an external merge tool should handle merging of multiple object types, a feasible, but time-consuming activity that results in a brittle tool that must be constantly upgraded as applications modify their object organizations. Alternately, the concurrency control technique can be set so that simultaneous development is impossible, at the cost of requiring serialization of editing activities. At present, no good solution exists.

Hypertext versioning capability is often added to an initially versioning unaware system, as was the case with Neptune/HAM, CoVer, the Hypermedia Version Control Framework, and WebDAV/DeltaV [31]. When this happens, there can be an existing base of versioning unaware clients that need to interact with the system even after versioning support is added. One technique for ensuring this is to carefully develop the interfaces in the API so that omitting version information (such as a revision identifier) causes useful default behavior to occur. The Hypertext Abstract Machine (HAM) used by Neptune provides a good example. When versioned composites were added to the HAM, it resulted in the addition of the notion of an active composite revision. Existing entry points in the HAM API still work, since they do not specify a specific composite, and hence they now use the active composite by default. Another way to handle versioning unaware applications is to automatically version items when they are written to the repository. In DeltaV, it is possible to turn on auto-revisioning for an object, meaning that every time it is written, it causes the DeltaV server to perform a check-out, write, and then check-in.

8.14 Interaction with an External Repository

In link server systems, the responsibility for storing representations of works is external to the hypertext system, which manages, and provides storage only for the hypertext network. When version control is added to both the external work objects and the internally stored anchors and links, the problem arises of synchronizing the external with the internal, across a control boundary.

The structure versioning design space (Section 8.5.2) can be applied here. Objects in the external repository cannot possibly be contained inclusively, since they are in a different repository, and hence they must be contained referentially. Furthermore, a revision selection rule over an externally stored versioned object cannot be evaluated by the link server, and must be evaluated by the external store. Additionally, it might not be possible to directly interact with the external repository; use of an intermediate store, such as the filesystem, might be necessary. Given these restrictions, it is possible to use the structure versioning design space to create a structure container that holds a subset of the versioned hypertext. An example of this is given in Section 9.2.2, which describes versioning for the Chimera link server system.

The difficulty introduced by the split in storage is how best to maintain consistency of the structure container. It is possible that an object could be removed from the external repository, and hence there can be revisions of the structure container that are no longer realizable, and links that are no longer traversable. If a structure container is checked out, and is being edited, a change in an external versioned object may cause an update in the structure container. However, how are these changes detected, and transmitted to the link server system? Ideally the external repository can be integrated with the link server so it sends event notification messages when relevant changes occur within the external repository. Based on these notifications, the link server can better maintain the consistency of the structure container. Alternately, the link server could make use of the repository's query interface, if available, and directly request information on the current state of the external store. This implies there would be a lag between a change in the repository, and its detection by the link server system.

8.15 Namespace Interactions

Discussion of how to design namespaces to satisfy the requirements from Section 7.14 (Namespace Interactions) is provided in Chapter 5 (Address and Name Spaces).

8.16 Maintaining Consistent Structures

If it is possible to revert a single revision of a work or link within a composite, it raises the issue of how to adjust the revisions of other links and works within the composite. If a work is reverted to a previous revision, then links in the composite selecting the original revision are no longer consistent with the new state of the composite. A traversal over one of these links would display a different revision from the one currently selected by the composite. Similarly, if a link is reverted, the endpoint revisions may no longer be the work revisions currently contained by the composite. In fact, the older link revision may be to a work revision no longer contained by the composite.

Revision consistency of a single link, L, within a composite, C, is defined as:

$$\text{link-consistent}(L, C) = \forall e_n, e_n \in L, \; e_1 \in C \wedge e_2 \in C \wedge \ldots \wedge e_n \in C$$

This states that, for all endpoints, e_n, of link L, the link is consistent only if composite C also contains all endpoints. Each endpoint is the identifier for a work revision, and resolution of the identifier yields V_n

for some work. Note that this definition only applies to links that have a work objects as endpoints. If the endpoints are anchors, the definition is:

$\text{link-consistent}(L, C) = \forall a_n, a_n \in L, e_n \in a_n, e_1 \in C \wedge e_2 \in C \wedge \ldots \wedge e_n \in C$

That is, for all anchors of link L, the link is consistent only if the object identifier part of each anchor ($e_n \in a_n$) is a member of the composite C. The definition of link consistency can be used to define consistency of the composite:

$\text{composite-consistent}(C) = \forall L_m, L_m \in C, \text{link-consistent}(L_1, C) \wedge \ldots \wedge \text{link-consistent}(L_m, C)$

A composite is considered consistent if all of its contained links are consistent.

If a new revision of a work or a link is selected in a composite, it may result in composite-consistent(C) evaluating to false. If an older work revision is selected, consistency might be re-achieved by reverting to older links having revisions of the work as an endpoint. It might be possible to find a link that has the older work revision, where the other endpoints are currently members of the composite. If an older link revision cannot be found, a new link revision can be made automatically, containing the currently selected revisions of the works at its endpoints.

If an older link revision is selected and causes the composite to become inconsistent, consistency might be re-achieved by selecting within the composite the work revisions at the endpoints of the link. However, changing these work revisions might cause additional links to become inconsistent. Creating new link revisions having the newly selected work revisions as endpoints resolves this inconsistency.

Chapter 9

Applying the Domain Model

In this chapter, the hypertext versioning domain model is applied to two systems, WebDAV/DeltaV [79, Clemm, 2000 #194], and the Chimera hypertext versioning proposal Whitehead, 1994 #17]. In both cases, the system went through a period of development, and was released, before versioning capability was added; the WebDAV protocol was fielded before the versioning extensions in DeltaV were fully developed, and the Chimera versioning proposal builds upon the versioning unaware capabilities of Chimera [7]. These systems all predate the hypertext versioning domain model, and hence the domain model was not used in the development of either WebDAV/DeltaV or the Chimera versioning proposal.

Since the addition of versioning capability to an existing system is the ideal scenario for application of the domain model, as described in the idealized process for application of the domain model in Section 1.3, it should be possible to use the domain model to develop a data model of these systems before and after versioning capability was added, and to reconstruct the original requirements, and the design choices chosen to satisfy them. By comparing their actual requirements (DeltaV and the Chimera versioning proposal both have their own, independently developed sets of requirements) with the ones in the domain model, and comparing the actual design choices against the design spaces listed in Chapter 8, it is possible to judge if the domain model is complete, and has sufficient descriptive power to handle the two example systems. The exhaustive examination of the domain requirements and design choices, along with the modeling of the complex data models of both systems, provides evidence that the domain model is complete.

In addition to its validation role, this chapter also serves as an example of the kind of outputs that would result from application of the domain model to add hypertext versioning to a versioning unaware system.

9.1 WebDAV and DeltaV

The Web Distributed Authoring and Versioning (WebDAV) protocol [79] is an extension to the Hypertext Transfer Protocol (HTTP) [68]. Though the detailed set of requirements for WebDAV have been captured elsewhere [175], a synopsis of protocol goals are:

- **Concurrency control.** To prevent overwrite conflicts, support is needed for concurrency control that works equally for all Internet content types.
- **Metadata storage.** To allow the recording of metadata (properties) on resources, such as bibliographic information, support is needed for reading and modifying metadata.
- **Containers.** To allow hierarchical containment, support is needed for creating and listing container objects. Also, where meaningful, existing operations should be extended to work on containers.

Since HTTP has no mechanism for representing per-user variants, the only concurrency control technique that has the required high genericity is whole resource locking (see Table 8), supported in WebDAV using the LOCK and UNLOCK methods. Metadata storage is provided by extending the HTTP object model from an all data to a data plus properties organization (using the terminology from Section 4.2). The PROPFIND and PROPPATCH methods support reading and writing properties. Since HTTP had no notion of containment, a container object was added, along with the MKCOL method for creating them. Listing a collection is performed using PROPFIND. An in-depth discussion of these features, and their rationale, is provided in [201]. A data model diagram for WebDAV is shown in Figure 31.

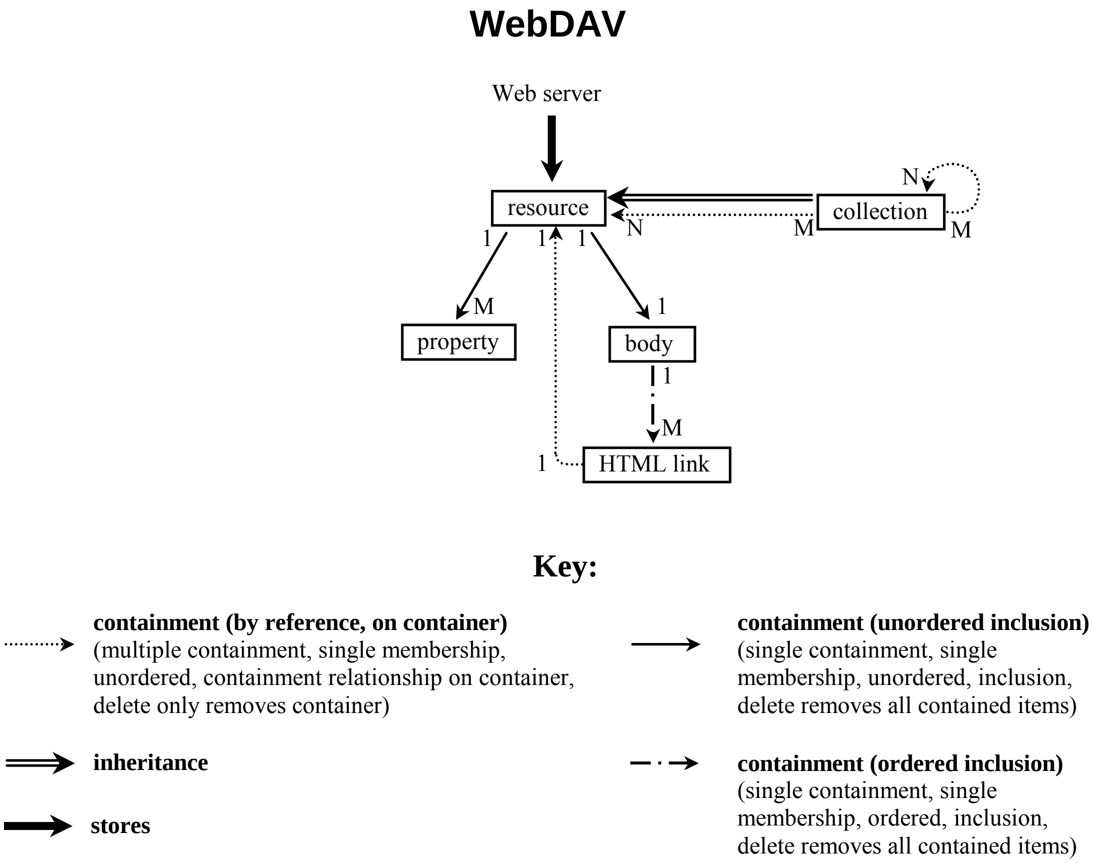

WebDAV
Web server
resource
collection
property
body
HTML link
N
M
1
Key:
containment (by reference, on container) (multiple containment, single membership, unordered, containment relationship on container, delete only removes container)
inheritance
stores
containment (unordered inclusion) (single containment, single membership, unordered, inclusion, delete removes all contained items)
containment (ordered inclusion) (single containment, single membership, ordered, inclusion, delete removes all contained items)

Figure 31 – The data model of the WebDAV extensions to HTTP [201]. The portrayed collection containment semantics are specified in the bindings extension to WebDAV [174].

Though the initial goals of the WebDAV protocol included support for version control (indeed, half of the requirements in [175] concern version control capabilities), in order to reduce the complexity of the protocol, and to ensure it would be completed more rapidly, version control capabilities are not in the WebDAV Distributed Authoring protocol [79]. Another working group, called DeltaV, was formed in the Internet Engineering Task Force (IETF) to extend WebDAV with versioning and configuration management capabilities.

9.1.1 Requirements

DeltaV has a rich set of requirements. Using the organization of the Domain Reference Requirements given in Chapter 7 as a guide, the following list highlights which domain requirements were, and were not, selected when adding versioning and configuration management capability to WebDAV. A complete list of goals for the DeltaV protocol are captured in the consensus goals document produced by the DeltaV working group [5], and a synopsis of these goals are presented in [199].

1. *The history of objects must be persistently stored.* This is a basic requirement of all versioning systems, and so DeltaV has this as a requirement.
2. *Immutable and mutable object revisions, and object metadata, must be supported.* In order to accommodate document management systems that allow checked in revisions to be modified, DeltaV does have a requirement to support mutable object revisions. The primary motivation is to support making small changes, and the preservation of logical revision names. However, this requirement directly contradicts the desire to support configuration management, which depends on revisions being immutable. As a result, mutable object revisions are only supported in the basic versioning feature set of the DeltaV protocol. Advanced versioning, which supports configuration management, does not permit mutable revisions.

   In DeltaV, properties cannot be versioned independently of their resource. This implies that client-settable properties can only be modified when the body of the resource is writeable. However, both WebDAV and DeltaV support "live" properties, where the server controls the state of the property. Live properties are often used for protocol-specific information, such as the revision set property of a history resource, which contains a URL for each revision in a versioned resource. For some live property values, the server may change the property value, even if its resource has been checked in, and is immutable. Additionally, there are some client-settable properties that must be writeable, even if the revision has been checked in immutable. Access control properties are the most important example, since it is desirable to be able to change the access permissions on a resource even after it has been checked in.
3. *Versioned and non-versioned objects can co-exist.* Remote software development is an important use scenario that DeltaV should support. Since the software build process often results in the creation of intermediate build objects (such as .o files created during the compilation of C code), there is a requirement that unversioned objects should be able to coexist in the same collections, and same portions of the URL namespace as versioned objects.

4. *All content types must be versionable.* Since the Web is composed of many different kinds of data, the DeltaV protocol has a requirement to provide version support for all content types, including those that are binary, rather than textual, in composition.
5. *A mechanism must exist for giving a human readable name to a single revision.* DeltaV has a requirement to support human readable revision labels, which can be set by a DeltaV client, to complement revision identifiers, which can are set by the server. DeltaV also has a requirement that these labels be unique only to a specific versioned object, that is, the same label value can be used in multiple versioned objects.
6. *Revisions and versioned objects can be removed.* To provide a mechanism that maps to the deletion or destruction capabilities of repositories that implement DeltaV, the protocol has a requirement to support deletion of revisions, and versioned objects.
7. *Stability of references.* DeltaV has a requirement to support linking to individual revisions. Since Web links are currently URLs embedded inside HTML and other document types (Word, PDF, etc.), this effectively requires each revision to have a URL, and this URL should be moderately stable. However, DeltaV does not have any explicit requirements concerning the stability of revision URLs, as this was considered to be too great an imposition on a server's ability to control and organize its URL namespace.
8. *Change aggregation support.* DeltaV has a requirement to be able to logically group a change, even though it may span multiple revisions across multiple versioned resources.
9. *It must be possible to version links.* At present Web links are URLs embedded within HTML, and other link-aware content types. Though work is progressing on an XML-based link standard, XLink [47], this standard has not yet been approved or adopted. Since all links are embedded within objects, by versioning objects DeltaV also versions links as a side-effect. However, there is no explicit requirement to version links in DeltaV, consistent with its emphasis on providing functionality similar to existing versioning and configuration management systems.
10. *It must be possible to version structure.* Since Web links are embedded in objects, it is impossible to separately version structure, and hence DeltaV has no requirement for this.

11. *It must be possible to link to a specific revision of an object.* As discussed above in #8, DeltaV does have a goal to support linking to a specific revision of an object.
12. *Variant support.* DeltaV has no explicit requirement to provide mechanisms for representing variants. However, DeltaV does explicitly require a branching version history, ostensibly to support parallel work, though this could also be used, in conjunction with labels, to represent variants.
13. *Collaboration support.* DeltaV has explicit requirements to prevent overwrite conflicts when multiple people work on the same revision at the same time. It also has a requirement to allow multiple simultaneous checkouts of the same revision. DeltaV also has a requirement to make it possible to prevent parallel development on a resource.

    Furthermore, though its goals document does not explicitly list this as a goal, it is certainly the intent that a mechanism be supplied that allows individual collaborators to work in isolation on the same set of objects, at the same time. This capability is provided by a workspace. DeltaV also has an implicit requirement to provide support for WebDAV locks, since DeltaV must support successful interoperation with versioning-unaware clients.

    DeltaV has an additional goal that it be possible to easily determine which resources have changed within a workspace. This makes it possible for collaborators to be aware of one another's changes.
14. *Navigation in the Versioned Space.* Besides providing the ability to link to a specific revision, the DeltaV protocol has no requirements for versioned hypertext navigation. In part, this is because DeltaV is an application layer network protocol, and hence is only concerned with interoperability between a client/server pair, and inherently has no responsibility for the client (or server's) user interface.
15. *Searching.* WebDAV does not have any search capabilities, beyond the limited ability of retrieving the values of a known set of properties from within a hierarchical set of collections. The DAV Searching and Locating (DASL) effort was created to address this shortcoming by providing the ability to remotely search a WebDAV repository [12]. In this context, DeltaV specified some additional requirements on searching capabilities:

    > "If the DAV server supports searching, it should be possible to narrow the scope of a search to the revisions of a particular versioned resource." [5], #12, p. 17.
    >
    > "If the DAV server supports searching, revision IDs and label names should be searchable." [5], #13, p. 17.

DeltaV also specifies some versioning-specific queries. Since a graphical display of a version graph is a common element of the user interface of many version control systems, it must be possible to retrieve, for a versioned resource, the complete set of revisions, their predecessor and successor information, the initial and latest revision, as well as label names. Furthermore, this information should be retrievable using a single network request so the information can be retrieved efficiently.

16. *Visualizing the Versioned Space.* Since DeltaV is a protocol, and not a client application, it has no visualization requirements.

17. *Traceability.* DeltaV has no requirement for tracing the (re)use of objects. DeltaV also has no real requirement, or support for change tracking, since activities, which support change aggregation, are not flexible enough, and do not support enough metadata, to provide a good change tracking system. Activities are limited to holding changes just on a single branch, and do not have a standard property, such as "change request identifier", that could be used to associate a set of changes with a human-entered description of the desired change. Activities could potentially be used in the construction of a change-tracking system, but alone they do not provide this support.

18. *Goals for User Interaction.* Since DeltaV is a protocol, and not a client application, it has no user interface requirements.

19. *Goals for Tool Interaction.* In many respects, since DeltaV is a network protocol, the entire focus of DeltaV is creating a standard interface for tools to interact with a versioning and configuration management repository. However, there are some more fine-grained requirements. Since DeltaV builds on the WebDAV protocol, and there are currently many commercially important WebDAV clients that are versioning unaware (e.g., Office 2000, Go Live 5), allowing versioning unaware clients to interact with a DeltaV server is important. DeltaV has a goal of providing version support for versioning unaware clients, automatically performing check outs and check ins, thereby creating new revisions.

    Tool adoption is another important issue. There is much concern that if the protocol is too complex, it will not be widely adopted. Furthermore, there are some repositories that only provide versioning operations, and have no configuration management support. For these two reasons, DeltaV has an explicit requirement to separate its functionality into two layers, basic and advanced. Basic support includes version control, while advanced features includes activities, workspaces, and configuration management. The version control layer provides all of the functionality most authoring clients need, and the advanced functionality can be employed by a more sophisticated client, such as a software development environment.

Since protocol requests issued by tools can potentially travel across the Internet, several security measures are necessary. First, users of tools will need to authenticate themselves, proving some evidence to the server that they are who they claim to be. Authentication in the reverse direction, a server authenticating itself to a client, is also desirable. Furthermore, it is also desirable to encrypt communications between client and server, so that any intermediaries are unable to eavesdrop. Finally, it might be desirable to encrypt resource contents on the server itself, providing some protection from unauthorized access there.

As a network protocol, DeltaV is likely to be used in non-English speaking countries. As a result, the protocol must provide support for encoding strings that will be read by a person so they can express all of the characters currently in use in human natural languages.

20. *Goals for Interactions with an External Repository.* DeltaV assumes that a server controls all of its versioned objects, and does not have any interaction with other repositories.

21. *Namespace Interactions.* DeltaV places several requirements on the URL namespace. Item #6 noted that DeltaV requires each revision to have its own URL. Since URLs do not have sufficient expressive power to annotate each URL path segment (text between "/" characters) with a revision identifier, forcing each revision to have its own URL effectively implies that the URL where a resource is authored will be different from the per-revision URL. For example, authorable URL "http://www.foo.com/myresources/index.html" might have a revision URL for revision 1.1 of "http://www.foo.com/repository/I/aa0011" or similar machine-generated URL.

    DeltaV also has a requirement that relative URLs should not be disrupted. That is, if a resource has a relative URL to another object in the same collection prior to being placed under version control, the relative URL should still work even after one or both objects are placed under version control. Effectively this means the namespace of objects should be the same before and after the objects are placed under version control. This is accomplished by replacing the unversioned resource with a versioned resource, which acts as a redirector for requests, forwarding them along to a specific revision. This leads to another DeltaV requirement, that each versioned object needs to have a settable revision that is returned for requests that do no explicitly identify a revision.

Since DeltaV was designed to be a protocol for accessing configuration management capabilities of remote repositories, it has several additional requirements that are not in the domain requirements for hypertext versioning systems.

1. *Baseline support.* A baseline is an object that records the specific revisions of all versioned resources specified by a workspace. That is, the workspace lists a number of versioned objects, and a specific revision for each versioned object. A baseline is a snapshot of the current state of a workspace. This capability directly supports configuration management.
2. *Policies.* DeltaV has a requirement that the protocol description should clearly identify the versioning and configuration management policies it dictates, and the policies that are still left to implementers and users.
3. *Location independence.* Since people sometimes change locations while they are working on a set of objects, for example, continuing work at home that was begun at work, it should be possible to start work using one client in one location, and continue that work using a different client in a different location.

There are several reasons why these requirements have not just been added to the domain requirements. While support for baselines provides some of the motivation for the domain requirement to maintain consistent structures (Section 7.15), baselines as separate from workspaces are in the realm of configuration management, and do not appear in hypertext versioning systems. No hypertext versioning systems have listed policy identification as a goal, and even the DeltaV protocol does not implement this requirement. Location independence is another goal that has not been identified by other hypertext versioning systems, perhaps because they have not addressed use by the same people from multiple locations. As more experience is gathered from the use of DeltaV, this requirement could conceivably migrate into the domain requirements; at present, it is left as a specialized requirement inherent to the remote access capabilities of DeltaV.

9.1.2 Design Choices

Satisfying the requirements for DeltaV led to the introduction of several new abstractions into the WebDAV data model, these being:

- *Revision, versioned object, history*: Together they represent the revision history of a resource.
- *Working resource*: A temporary, writeable resource created by a checkout, and converted to a revision upon a check in.
- *Activity*: Represents a logical change that can span multiple revisions of multiple versioned resources.
- *Workspace*: Used to create work areas for individuals or groups that are isolated from other collaborators.
- *Baseline*: Used to represent a configuration, and contains one revision each from the set of versioned resources within a workspace.

The data model for DeltaV, including these abstractions, is shown in Figure 32. Details on how DeltaV satisfies its requirements are given below, organized by requirement. The set of requirements below does not include all of the hypertext versioning domain requirements, since DeltaV does not share them all.

1. *Persistent storage of revision histories.* DeltaV employs a variant of the versioned objects approach for persistently storing object histories. Each revision is a separate object, and each revision has a property, predecessor set, that holds references (using URLs) to each of the revisions that precede it. Thus, the predecessor relationships are stored using the "containment relationship on container" containment type, where the container object is the revision itself. Unlike the typical versioned object approach, in DeltaV the versioned object is really two abstractions, the versioned resource, and the history resource. The versioned resource acts as a proxy for the complete version history within the authorable URL namespace. So, for example, if a resource existed at URL http://www.foo.com/index.html before being placed under version control, once it was versioned, a versioned resource would now exist at that URL. The contents of the resource prior to version control are now within a new revision resource.

   However, the versioned resource does not directly contain revisions. Instead, the versioned resource contains a history resource (via a URL reference stored in the history property) that has the responsibility for recording all revisions in the version history. The motivation for separating the versioned resource from the history resource is locking, and the need to reconcile it with the branching concurrency control supported by versioning. Versioning unaware clients, such as those that understand WebDAV, but do not understand the DeltaV protocol, only have locking available as a concurrency control technique. When interacting with a versioned part of the URL namespace, one where objects have been placed under version control and a versioned resource is available at each URL, a locking client would end up locking the versioned resource. If the versioned resource contained all of the revisions, only the lock owner would be able to perform a checkin, an operation that modifies the state of the versioned resource. By separating revision containment from namespace participation, the versioned object can be locked without limiting checkins.

2. *Revision mutability, and immutability.* Mutable revisions are controlled by a revision property called mutable. If the mutable property is true, then it is possible to overwrite the revision. WebDAV, hence DeltaV, properties are divided into dead and live properties, where dead properties have values set by the client, and live properties are set by the server, and hence the mutability of properties is divided into dead and live behaviors. Dead properties have the same mutability as the resource itself. Live properties are always mutable.
3. *Coexistence of versioned and unversioned objects.* This requirement is satisfied by the lack of any constraint on the membership of versioned containers, and by the division of container containment relationships into "public" relations that are permanently versioned, and "private" relations that are unversioned. While it is possible that versioned resources can be contained using a private relationship, the intent is for versioned objects to be contained using public relationships.
4. *Versioning of all content types.* This requirement is achieved by not introducing any restrictions on the types of content that can be versioned, and by not making any design decisions that depend on the content type being known. Thus, DeltaV is precluded from using within-object versioning or within-object variance techniques for all object types. Additionally, the existence of both binary and textual content increases the requirements on DeltaV repositories, since they need to support delta storage for all content types.

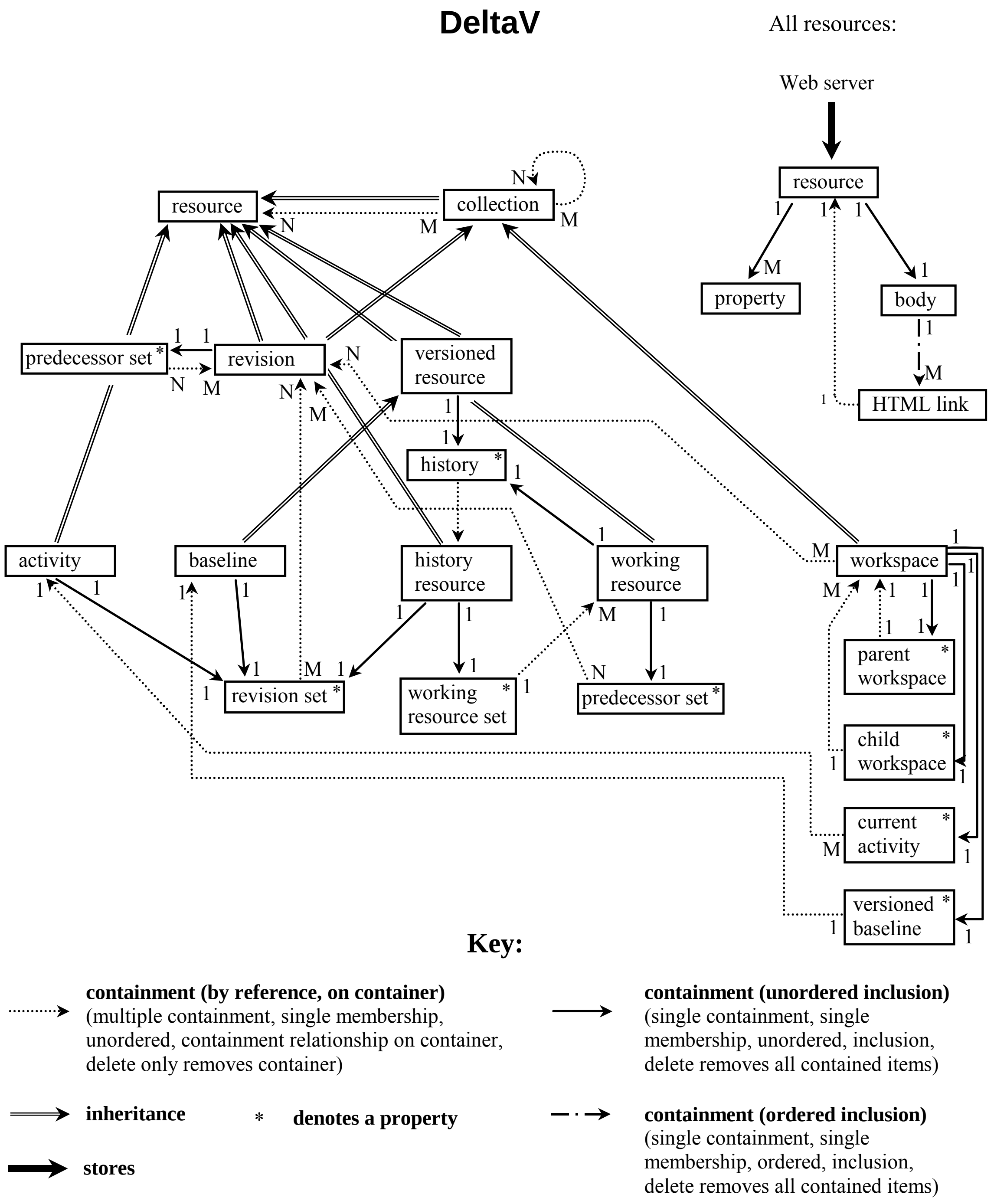

DeltaV
All resources:
Web server
resource
property
body
HTML link
resource
collection
predecessor set *
revision
versioned resource
history *
activity
baseline
history resource
working resource
workspace
revision set *
working resource set *
predecessor set *
parent workspace *
child workspace *
current activity *
versioned baseline *
N
M
1
Key:
containment (by reference, on container)
(multiple containment, single membership, unordered, containment relationship on container, delete only removes container)
containment (unordered inclusion)
(single containment, single membership, unordered, inclusion, delete removes all contained items)
inheritance
* denotes a property
containment (ordered inclusion)
(single containment, single membership, ordered, inclusion, delete removes all contained items)
stores

Figure 32 – Data model of the DeltaV versioning and configuration management extensions to WebDAV. The data model is based on revision –05 of the DeltaV protocol specification [31], and depicts advanced versioning. Many properties are not shown on this diagram to reduce visual clutter. Properties shown in this diagram represent containment relationships between entities.

5. *Revision naming.* DeltaV provides a facility called labels that allow a human readable name to be attached to a revision.
6. *Removing revisions and versioned objects.* Though DeltaV identified deletion of revisions as a goal, the protocol specification does not mandate support for this feature, leaving the results of a delete on a revision “undefined,” meaning that it is possible a server supports this capability, though a client cannot depend on it being present. Similarly, the effect of a delete on a versioned resource is undefined, except in the case where the versioned resource is contained by a versioned collection, in which case no deletions are allowed. Due to the lack of definition, and the contradiction with the stated requirements, this capability will likely be clarified before the DeltaV protocol is completed.
7. *Change aggregation support.* DeltaV introduces the activity object to act as a container for logically related changes. However, since the activity is a subtype of a plain resource, it is not a WebDAV/DeltaV collection, and hence the activity is an example of the “new container” approach discussed in Section 8.4. The activity contains the set of related revisions within a property called the revision set. An activity is associated with a working resource on checkout, so that when it is checked in, the activity will correctly add the new revision.
8. *Linking to a specific revision.* DeltaV mandates that every revision have a separate URL, called a revision URL, that can be used to directly access it. This revision URL is also used as a revision reference for containers like the history resource, activity, and baseline.
9. *Collaboration support.* Since DeltaV needs to provide equal versioning support for all content types, it is limited to selecting from concurrency control techniques that have high genericity. Since support for parallel work is also a requirement, whole-resource locking, as provided by WebDAV, is insufficient. The only concurrency control technique left that has high genericity is branching; hence DeltaV uses this technique.

When a resource has been checked out, it must be writeable, allowing changes to it. In DeltaV, though clients will likely perform local, client-side caching of resources that are being authored, this behavior cannot be depended on, and hence there is a need to provide a writeable resource on the server that can be written to using the standard PUT method. The writeable resource created during a checkout is a *working resource*. Since multiple checkouts can be performed on a single revision, the working resource directly supports branching concurrency control.

DeltaV also has a requirement to provide isolated work areas for individuals, or small teams. This requirement is satisfied by the introduction of workspaces. Since multiple working resources can simultaneously have the same revision as their parent, the workspace is used to select which of these working resources a particular person authors. A workspace also provides revision selection, picking a specific set of revisions for a set of versioned objects. This allows people to simultaneously work on different views of the same set of resources, supporting tasks such as performing a bug fix on an older, released version of a software project, or making changes to a set of documents without having to constantly integrate the changes other collaborators. Unlike the composite-based hypertext versioning systems where workspaces and collections are combined, DeltaV separates the workspace, providing isolated work areas, from collections, used generally for grouping resources.

DeltaV provides the ability to merge together two collaborator's work once they have finished working in parallel. Since DeltaV cannot guarantee it will understand the internals of every content type, it cannot provide automatic merging for resources. Furthermore, automatic merges may produce semantically incorrect results. When the MERGE method has been invoked, DeltaV checks out resources that are in conflict, and flags them by setting the "merge-set" property on the working resources created by the check out. Unless the server has specific knowledge about the internal organization of the content type allowing it to perform an automatic merge (as might be the case for plain text), the client is then responsible for performing the actual content merge, presumably by creating a display so a user can select changes from each revision to preserve. Activities can be used to select which revisions to merge, thus allowing one user to include a logical change made by another collaborator into their workspace.

10. *Navigation in the Versioned Space.* Though DeltaV does not have a requirement for supporting navigation through a consistent time slice, by using workspaces, and default revisions, this capability is supported fairly well. Since workspaces select revisions from a set of versioned resources, workspaces can be used to select an internally consistent set of revisions, which can then be navigated among.

11. *Searching.* DeltaV provides a mechanism for generating versioning-specific reports, using the REPORT method. Provided reports include:

    - A version history report,
    - A successor report for a revision, needed since each revision only stores predecessors,
    - A checked out report that lists all working resources derived from a specific revision,
    - A latest checkin report that lists the most recently checked in revision within a versioned resource,
    - A "property report" that takes advantage of the fact that many properties are acting as collections (e.g., revision set, predecessor set), and allows properties to be retrieved from the members of these "property collections."
    - A merge preview report, detailing what would happen if a set of revisions were merged into a workspace, without actually performing the merge,
    - A compare report that allows the membership of two baselines or workspaces to be compared.
    - A current workspace report, which identifies, for a specific activity, those workspaces where work is being performed on the activity.

12. *Goals for Tool Interaction.* As noted above in Section 9.1.1, item #19, DeltaV separates its functionality into two layers, basic and advanced. The purpose of this separation is to reduce the adoption barrier for clients and servers that only require version control capability.

    DeltaV relies on HTTP and WebDAV for its authentication mechanisms. HTTP provides two forms of authentication, Basic and Digest [73]. Basic authentication effectively sends username/password pairs in the clear, while Digest authentication performs multiple one-way hashes on the username/password pair before it is sent over the wire. Encryption of the communication path between client and server is accomplished using Secure Sockets Layer (SSL), also known as Transport Layer Security (TLS) [79]. This protocol prevents casual eavesdropping of data transmitted between client and server.

    For strings that will be presented to a human in a user interface, DeltaV relies on the internationalization capabilities of the Extensible Markup Language (XML), which is capable of recording both the language, and the characters of most human languages [24]. DeltaV employs XML for marshalling of protocol information. This information can be used to correctly display these strings to a user.

13. *Interactions with the Object Namespace.* As discussed above in Section 9.1.1, item #21, DeltaV requires both that the original URL namespace be preserved, so that relative URLs still behave correctly for objects under version control, and that each revision has its own URL.

Additionally, DeltaV satisfies some requirements that are not part of the hypertext versioning domain reference requirements.

1. *Baseline support.* Baselines are an additional container-like object added into the DeltaV data model to permanently record a snapshot of the revisions of resources and collections comprising a project. In DeltaV, each baseline is a versioned resource, and is associated with a specific workspace. When a baseline is checked out, and hence writeable, its revision set property contains the same set of revisions as are contained by its associated workspace. When a baseline is checked in, and hence immutable, it contains a snapshot of the contents of the workspace as of the moment of checkin. A baseline can also be used to populate a workspace with a set of revisions. This allows a collaborator to start working from a known configuration of resources, such as a released, or tested version of a project.
2. *Policies.* Though the DeltaV protocol has several use processes deeply embedded within it, these are not discoverable via the protocol, and at times are even implicit in the protocol description itself. Thus, the goal of making policies explicit and discoverable has not been met.
3. *Location independence.* The goal of accessing a DeltaV repository equally from multiple locations has largely been met just by using HTTP as a base for extension. HTTP is a stateless protocol, implying that there is no protocol state associated with a specific network connection. Hence, even if someone is working against a DeltaV repository from work, they are still able to perform operations on the same set of resources from another location, even if the DeltaV client is still running at work. The only potential difficulty might arise when using only WebDAV aware clients against a DeltaV repository, since these clients would be using locking for concurrency control, and thus might prevent the same user from accessing a resource, unless the client is intelligent enough to retrieve and resubmit lock tokens.

9.2 Chimera

Chimera-1 [7] provides hypertext services to the heterogeneous tools and data sources populating software development environments. Chimera manages a hypertext structure of anchors and links over the various works created during software development, such as source code, specifications, design and requirements documents, etc. Since storage of these artifacts is under the control of applications, Chimera is classified as a link server system [142]. The separation of storage responsibility between hypertext and artifact storage allows Chimera to create hypertext webs over works stored in multiple repositories, from the filesystem to an application-specific store.

Chimera's abstractions can be divided in two groups, those used to create hypertexts, and those that model the external environment of tools and works. A Chimera link is defined as a set of anchors, and anchors are defined on symbolic renditions, called *views* by Chimera. A symbolic rendition is created by a renderer, in this case a specific application in the environment, termed a *viewer* by Chimera. The renderer operates on a data representation of a work stored in an accessible repository; Chimera uses the term *object* to refer specifically to the representation of a work, a more restrictive definition than the definition of object as a single or aggregate data item, in use to this point. Chimera models viewers and objects as references out into the environment. A view is modeled as a *(viewer, object)* pair. The anchor can be viewed as the connection between Chimera's hypertext abstractions, and its model of the external environment. Figure 33 displays Chimera's data model.

Figure 34a shows the architecture of the Chimera system. Applications in the environment that desire access to Chimera's hypertext services have been integrated using Chimera's API so they communicate with the Chimera server, thus making them clients. The Process Invoker component executes applications that are needed to complete a link traversal. Figure 34b shows how a Chimera client creates a rendition that includes hypertext anchors and links. The client reads an object from its native repository, and combines that with anchor and link information retrieved from the Chimera server to create its symbolic rendition. If the user initiates a link traversal, the client communicates the link traversal to the Chimera server. The Chimera server checks to see if any additional clients need to be running to complete the link traversal. If so, it instructs the Process Invoker to start the needed client application(s). Once all the applications are

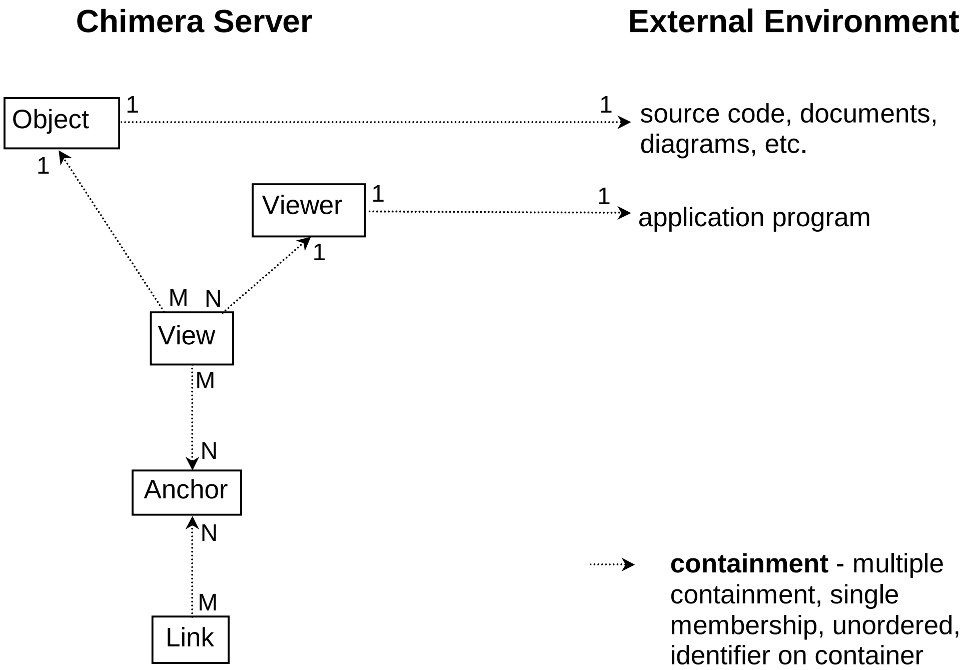

Chimera Server
External Environment
Object
1
1
1
source code, documents, diagrams, etc.
Viewer
1
1
1
application program
M
N
View
M
N
Anchor
N
M
Link
containment - multiple containment, single membership, unordered, identifier on container

Figure 33 – Chimera data model, with no versioning support.

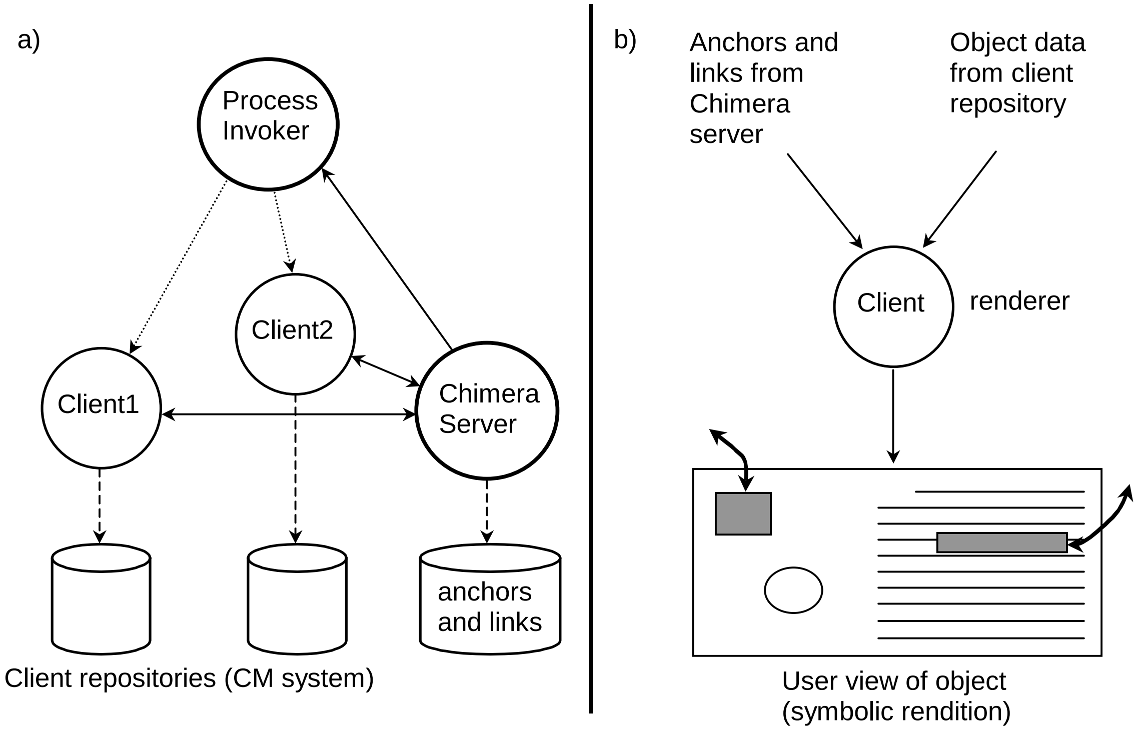

a)
Process Invoker
Client2
Client1
Chimera Server
anchors and links
Client repositories (CM system)
b)
Anchors and links from Chimera server
Object data from client repository
Client
renderer
User view of object (symbolic rendition)

Figure 34 – The Chimera-1 architecture, and the creation of a symbolic rendition by a client application.

running, the Chimera server sends them link traversal messages, causing them to display their link endpoint.

Software development environments typically employ either a version control or configuration management system to record revisions of works as they are developed. Unfortunately, Chimera does not handle versioned objects well. If an object is reverted to an older revision, Chimera has no way of ensuring that anchors within views of the revision will remain attached to right words or symbols, since Chimera only stores anchor positions for the latest revision. Furthermore, Chimera has no way of tracking changes to links over time, and thus cannot version structure.

To address these shortcomings, a proposal for versioning support for Chimera was developed in 1994 [200]. For the sake of brevity, the discussion below will used the name ChimeraVP to describe the Chimera system as amended with the Chimera versioning proposal. The following sections describe the requirements that guided this proposal, and the recommended approach for meeting the requirements. Unfortunately, this proposal was never implemented. However, it is interesting to examine since it is the only known work that addresses structure versioning where works and their revisions are outside the control of the hypertext system, and thus provides an interesting test case for the application of the hypertext versioning domain model.

9.2.1 Requirements

Using the organization of the Domain Reference Requirements given in Chapter 7 as a guide, the following list describes the requirements for ChimeraVP. While the proposal itself listed several requirements, they are a subset of the requirements below. Since the Domain Reference Requirements were developed by looking at a broad range of systems, it is not surprising that they are more comprehensive.

1. *The history of objects must be persistently stored.* Within the Chimera system, the term object refers to just the representation of works, however, this requirement applies to all abstractions. For ChimeraVP, responsibility for maintaining work revisions lies with the external, heterogeneous repositories in the environment, and hence outside the control of ChimeraVP. However, ChimeraVP does have a requirement that it must accommodate work revisions, if present. ChimeraVP also requires that all

abstractions under its control, both the hypertext abstractions (anchor, link) and the abstractions used to model the external environment (view, viewer, object) are versioned.

2. *Immutable and mutable object revisions, and object metadata, must be supported.* ChimeraVP does not have any requirements concerning the mutability of work revisions, since these are outside its scope. For its internal objects, ChimeraVP requires revisions of their primary state to be immutable, so structure versioning can be performed. However, since Chimera stores access permissions on attributes, minimally at least access permission attributes must be mutable, even on checked-in revisions.
3. *Versioned and non-versioned objects can co-exist.* ChimeraVP requires that its internal objects must all be versioned, and hence does not support this requirement. However, the primary motivation for this requirement is to allow versioned and unversioned *works* to co-exist, and here ChimeraVP has no requirements, since this is a property of the external store, outside its control.
4. *All content types must be versionable.* Since software development environments contain a wide variety of work content types, ChimeraVP definitely has the support of all content types as a goal. However, for ChimeraVP an external repository performs the versioning of works, and hence this requirement translates into a constraint on these external repositories.
5. *A mechanism must exist for giving a human readable name to a single revision.* The original system described in the Chimera versioning proposal did not have support for revision selection rules, and hence also had no support for labels, or other human readable revision names. However, even if revision selection support is not provided, it seems reasonable to list human readable name support as a goal, so that specific link revisions, for example, could be given meaningful names. An application program could potentially use this facility to consistently label anchor and link revisions with the same label used for work revisions, for example, with the name of a software release.
6. *Revisions and versioned objects can be removed.* Since external repositories can potentially delete revisions, or entire revision histories, ChimeraVP must be able to accommodate this. Additionally, to ensure scalability over time, ChimeraVP should be able to remove old, unused (or infrequently used) revisions of objects.

7. *Stability of references.* A basic architectural decision in link server systems is the ceding of control over the storage of works and their revisions. Due to this, works may be deleted or modified at any time, and the link server cannot prevent this. As a result, reference stability cannot be guaranteed by a link server system alone; it must be accomplished by the cooperation of the link server and the external repositories. As a result, ChimeraVP does not have absolute stability of references as a goal. However, if a work is merely changed, and not deleted, then ChimeraVP has a requirement that anchors must accurately track their endpoints over work revisions. As [200] states, "hypertext versions will get out of synchronization with object versions – a hypertext versioning system must accommodate this." (p. 46)

8. *Change aggregation support.* ChimeraVP assumes that change aggregation support is a feature provided by external repositories, and hence this is not a requirement for ChimeraVP.

9. *It must be possible to version links.* Chimera links are defined as a set of anchors, and this set of anchors can change over time. Additionally, Chimera links have attributes defined on them, and this metadata can change over time as well. As a result, ChimeraVP must provide link versioning support, so that individual link changes can be tracked.

10. *It must be possible to version structure.* ChimeraVP requires that the link structure centered on a specific work revision must be versioned. That is all links that have an anchor on a view of the work revision are included in the structure. This structure centered on a work revision is necessary to select the revisions of Chimera objects that correspond to a specific work revision. When a client application contacts ChimeraVP, it typically only knows the name and revision of the work it is rendering, and ChimeraVP must to associate this work revision with the Chimera object revisions (viewer, object, view, anchor, link) that correlate with that work revision.

11. *It must be possible to link to a specific revision of an object.* ChimeraVP supports this requirement. Since an application program can request the creation of an anchor on any revision of a work, ChimeraVP must be able to accommodate this, and the inclusion of these anchors in links.

12. *Variant support.* Since many versioning and configuration management systems support variant segregation (see Section 8.6.1), often using branches of a revision history, ChimeraVP must accommodate this behavior.

13. *Collaboration support*. While collaboration support was not explicitly addressed in the original Chimera versioning proposal, since multiple developers usually work simultaneously on large software projects, collaboration support is a must. This implies sub-requirements for concurrency control to avoid overwrite conflicts on Chimera objects, and merge support to combine multiple simultaneous changes. The ability to work in isolation is also desirable, requiring a combination of work isolation capabilities in ChimeraVP and the external repository.

14. *Navigation in the versioned space.* ChimeraVP has a requirement to support navigation that is linkwise consistent. That is, the endpoints of a specific link should be consistent with the meaning of the link. However, traversal of any other link is not guaranteed to result in a rendition of a work revision that is consistent in time (or any other measure of consistency). The rationale for this behavior is the assumption that the facilities of the external repository will be used to create consistent collections of works, such as by using configuration management facilities. By focusing on versioning the structure centered on a single, the external repository has maximum flexibility for creating arbitrary compositions of work revisions.

    Support for navigation from revision to revision must be supported by ChimeraVP, although the actual user interface for initiating such a link traversal must be supplied by a specific application, and adds to the integration effort. Additionally, the external repository must support the retrieval of an arbitrary work revision (see requirement #19 below).

15. *Searching.* ChimeraVP has no requirement for supporting revision selection rules. Additionally, since works are externally stored, searching work revisions must be performed in each repository. The decision not to support revision selection rules is fairly arbitrary. Not providing this feature results in a simpler system, while their presence would increase the flexibility of revision selection.

16. *Visualizing the versioned space.* ChimeraVP, being an infrastructure service with a minimal user interface, does not itself have any requirement for visualizing the versioned space. However, ChimeraVP should provide sufficient capabilities in its API so that a hypertext viewer client could be constructed. Such a viewer could provide a visualization of a versioned hypertext.

17. *Traceability.* ChimeraVP tends to assume that works are complete, and not parts of some composite, and anyway treats the reuse of composite parts as something outside its control, a concern of specific

applications and repositories. As a result, ChimeraVP has no requirement for tracing the reuse of parts of a work.

18. *Goals for user interaction.* This requirement is explicitly noted in [200], "The versioning of hypertext structures should occur as transparently to the user as possible. The versioning system should not constantly prompt the user for version names, instead automatically selecting and assigning version names where possible." (p. 46)
19. *Goals for tool interaction.* Several tool interaction goals are listed in [200], p. 46-47. First, it lists as a goal, "demands on external tools should be minimized." Next, it states that an "external repository must be able to retrieve an arbitrary version of an object," so it is possible to complete a link traversal to an arbitrary work revision. If the external repository makes work revisions available via the filesystem (a common feature in SCM systems), an application must have some way of determining the current revision of a file.
20. *Interaction with an external repository.* This goal is explicitly listed in [200], p. 46. "Objects stored external to the hypertext system and hypertext structures stored internal to the hypertext system must be capable of independent development." Support for this goal has affected almost every requirement.
21. *Namespace interactions.* Naming of work revisions stored in external repositories is outside the scope of ChimeraVP. ChimeraVP requires simply that applications be capable of distinguishing the name associated with the abstract work concept, or versioned object, from the identifier or name used for individual work revisions. Applications must be able to communicate this information to ChimeraVP, since it is used as the key for determining the subset of Chimera's structure centered on a specific work revision.
22. *Maintaining consistent structures.* SCM systems often provide the ability to revert to a previous revision of a single work, without also reverting all other works in a given collection, or workspace. As a result, ChimeraVP must be flexible enough to accommodate this behavior.

9.2.2 Design Choices

Satisfying the requirements resulted in the addition of the time dimension by making all Chimera object, so that they would be capable of versioning links, anchors, as well as changes to the abstractions used to model the external environment. For this latter set (views, viewers, objects), the addition of versioning is primarily intended to capture changes in attributes, though the ability to capture changes in the name of works (the value of a Chimera object) is also useful.

Several new abstractions were added to the Chimera data model to satisfy the requirements:

- *Configuration.* A configuration holds a subset of a hypertext containing all of the Chimera objects needed by an application rendering a specific work revision. So, for each work revision, it holds a revision of a viewer, object, and view, along with all anchor revisions defined on the view, all links revisions containing those anchors, and all additional anchor revisions contained by the links. Configurations are versioned, with the intent being that each revision of a work will have a corresponding configuration revision.
- *Version association table.* The version association table creates a bi-directional association between a *(work identifier, revision identifier)* pair, such as a filename with a revision identifier, with a specific configuration revision. The version association table acts as the bridge between the versioned repository and a versioned hypertext structure.
- *Versioned objects.* For each Chimera object, a versioned object is introduces to be a container for its revisions. So, a versioned view, versioned viewer, versioned object, versioned link, and versioned anchor were all added to Chimera’s data model.

The data model for ChimeraVP, including these new abstractions, is shown below in Figure 35. An example of this data model is shown in Figure 36.

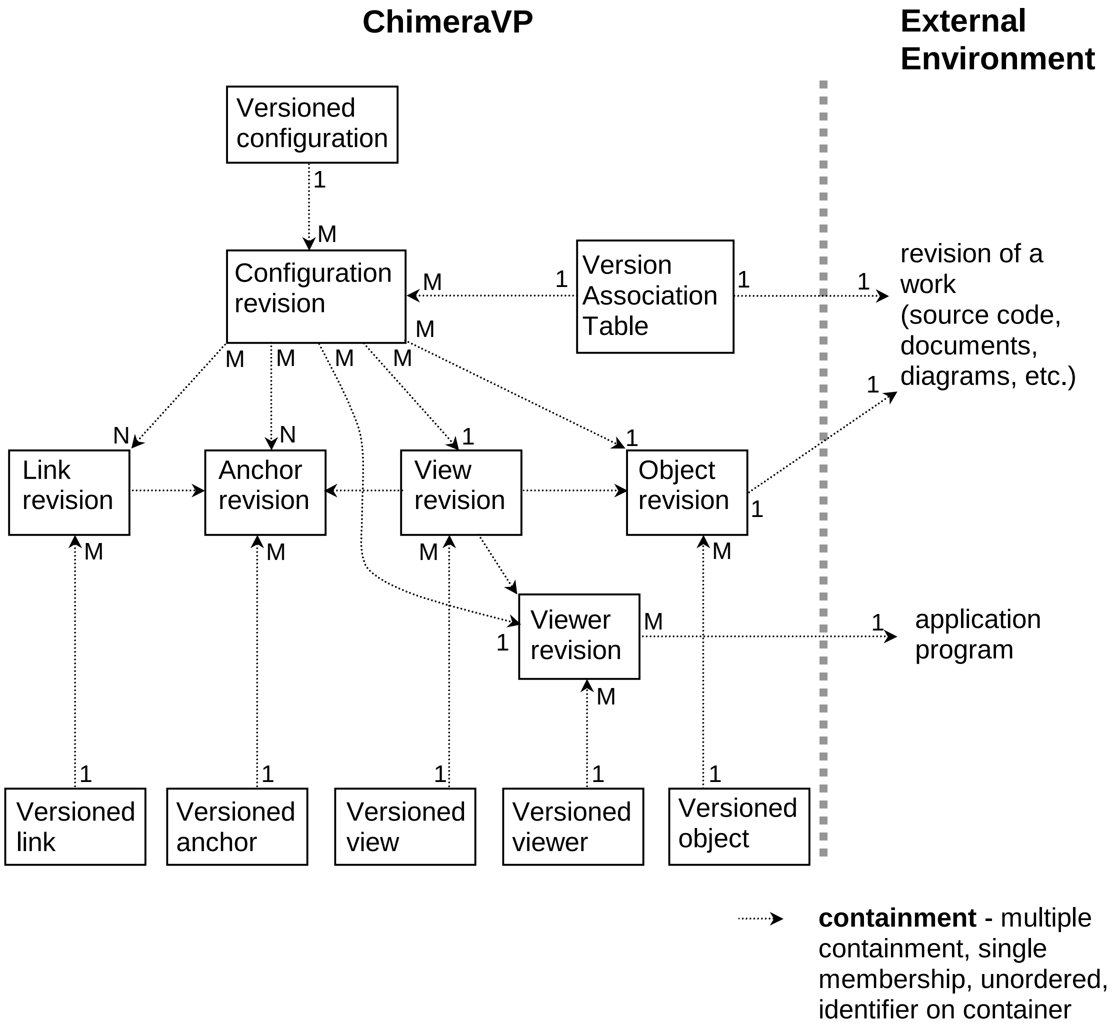
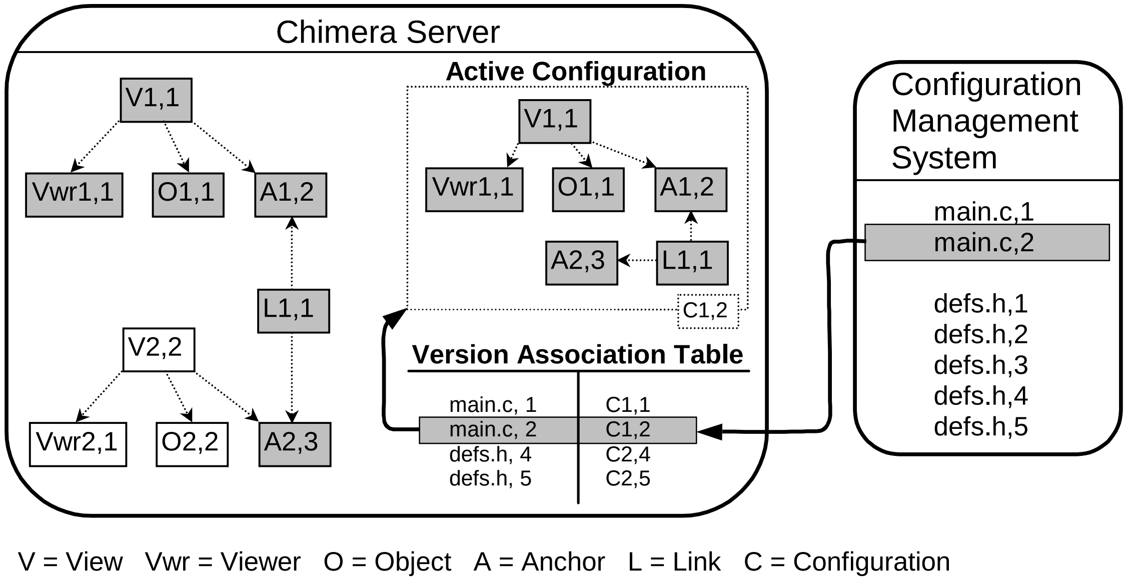


Figure 35 – Data model of the Chimera system, with version support added.



main.c, 1	C1,1
main.c, 2	C1,2
defs.h, 4	C2,4
defs.h, 5	C2,5

Figure 36 – An example configuration, showing the path from the configuration management system to the configuration. Shaded items all pertain to the same configuration revision.

Details on how the Chimera versioning proposal addressed the requirements are given below, organized by requirement. The sections below cover a subset of the Domain Reference Requirements, since ChimeraVP does not use them all.

1. *Persistent storage of revision histories.* ChimeraVP uses the versioned object approach for persistent storing the revision history of all its objects. This is visible in Figure 35, which shows every Chimera object with its matching versioned object. Since revisions can only belong to a single revision history, the versioned object stores the predecessor/successor information. Repositories in the external environment are responsible for versioning works, and may use a variety of techniques for doing so; ChimeraVP can accommodate any existing versioning technique, as long as the external repository requirements are satisfied (described above, in item #19).
2. *Revision mutability, and immutability.* The mutability of work revisions is under the control of external repositories. While the mutability of the attributes of Chimera objects was not discussed in the original Chimera versioning proposal, an approach that would satisfy the need for mutable attributes is to have a mutability flag that is settable when the attribute is created. Predefined attributes, such as those used for access control, would have their mutability predefined by the Chimera system. The actual contents

of the Chimera objects is only mutable when performing consistency cleanup after the destruction of anchor and link revisions.

3. *Versioning of all content types.* Since ChimeraVP does not provide storage for works, this requirement must be satisfied by ChimeraVP's accommodation of a wide variety of external repositories. However, ChimeraVP also satisfies this requirement by not making any assumptions about the content type of works, assuming them to be opaque objects.
4. *Revision naming.* ChimeraVP can easily support this requirement by having applications store a label in an attribute on the Chimera object, and by establishing a convention by which a standard name is used for this attribute, such as "label."
5. *Removing revisions and versioned objects.* External repositories control whether work revisions can be deleted or destroyed. When a work revision is destroyed, ChimeraVP should remove its associated configuration revision, and its entry in the association table, to free up its storage. Additionally, to aid the scalability of the ChimeraVP repository, it should be possible to destroy old, unused revisions of Chimera abstractions (anchors, links, etc.). Destruction of these revisions results in a consistency update step, where references to the deleted objects are removed.
6. *Stability of references.* The complete design space for ensuring link endpoint consistency in open hypertext systems is described in [41]. At present, ChimeraVP requires that applications must inform the Chimera system when changes are made to anchor endpoints in order for Chimera to correctly update the anchor endpoints. Since previous revisions are available, more sophisticated techniques are possible. For example, Chimera could grab the surrounding context of an anchor from a previous revision, then search for that context in a later revision, updating the anchor if found. However, this search and update capability would need to be added to individual client applications, since only they have knowledge concerning the internal organization of work objects.
7. *Link versioning.* ChimeraVP versions links using the versioned object approach. Anchors are also versioned in this way, and hence links in ChimeraVP contain a set of anchor revisions.
8. *Structure versioning.* The configuration, and version association table abstractions were introduced to provide versioning of the structure associated with a single work revision. This can be viewed as an instance of the structure versioning design space. The structure container is the configuration, and it

holds one Chimera viewer, object, and view revision, along with any number of anchor and link revisions. All containment relationships are referential, with multiple containment, single membership, no ordering, and the containee identifier stored on the container. No revision selection rules are in use.

The version association table provides the mapping between a revision in an external repository, and a specific configuration revision, representing the subset of the Chimera database that applies to it.

9. *Linking to a specific revision*. Support for this requirement is an emergent property of ChimeraVP's design. Since any revision of a work can be associated with a configuration, and since an anchor can be defined on any work revision, it is possible to create a link to a specific work revision.

10. *Variant support.* If an external repository uses variant segregation to model variants, meaning that each variant is a separate object, then ChimeraVP can provide variant support as-is, so long as each work variant is given a unique identifier that can be combined with its name to create the *(name, identifier)* pair that is passed to the version association table. Support for within-object storage of variants is also possible, so long as the external repository provides a mechanism for extracting or working with specific variants.

11. *Collaboration support*. For collaboration support, ChimeraVP needs to provide concurrency control to avoid the lost update problem. Concurrency control for objects is the responsibility of applications, and is outside the control of ChimeraVP. ChimeraVP can provide concurrency control items in its repository. Since these items typically have small amounts of state, and the Chimera server creates a single point of control, whole object locking is a good form of concurrency control. Locking is frequently used by SCM systems, and so this choice is consistent with the typical concurrency control mechanism used by external repositories.

    When locking, the ChimeraVP server should lock only a subset of the items in a configuration, specifically the viewer, object, view, and all anchors defined on the view. Links should not be locked, nor should anchors defined on other views. Since links span views, a policy of locking links would prevent collaborators from working on views reachable by links from an active view.

12. *Navigation in the versioned space.* Introduction of the configuration satisfies the requirement to support linkwise consistent navigation. It is possible for applications to create revision-to-revision

links by creating an anchor revision on a view of each work revision, then creating a link holding both anchor revisions.

13. *Visualizing the versioned space.* The information needed by an external application to visualize the versioned space—the membership lists of configurations, links, and views—is the same information needed by applications integrating with ChimeraVP. As a result, creating an API usable by client applications satisfies this requirement.
14. *Goals for user interaction.* Minimizing the need to ask the user for revision identifiers and names can be addressed by the ChimeraVP system automatically assigning revision identifiers. This will reduce the need for applications will constantly ask the user for this information. However, since the primary user interface to ChimeraVP is through applications, it is ultimately their responsibility to meet the user interaction requirements.
15. *Goals for tool interaction.* While the addition of version control capability will lead to additional burdens on tool interactions with the Chimera server, aspects of the Chimera versioning proposal act to minimize these additions. For each user, ChimeraVP supports a series of active configurations, one for each work revision currently being edited or viewed by an application run by the user. Since the active configuration contains all of the ChimeraVP object revisions used by the application, it is possible for the application to omit revision identifier information, using the revision contained by the active configuration instead. This has the nice quality that existing applications which are versioning unaware still have the ability to interact with a versioned ChimeraVP repository, thus reducing the burden on tool integration. This is similar to the approach used by Neptune/HAM when versioning was added to contexts; the existence of a default context allows tools to default the revision identification in their interactions with the HAM [46]. Of course, such an application would be unable to provide information to the user concerning the revision of link destinations, or be able to create links across different revisions of the same work.
16. *Maintaining consistent structures.* The configuration is the key to ChimeraVP's support for the ability to arbitrarily revert a single work revision in a collection or workspace. A configuration holds only that part of the hypertext structure centered on a single work revision, and limits the enclosed items to just the endpoint anchors of all links emanating from the work's rendition. This is much less than the

hypertext subset held within the composites used by composite-based systems. Due to the focus on a single work revision, it is easy for ChimeraVP to select another configuration revision when a work revision is changed. In contrast, when a work revision is reverted in a composite, the consistency issues discussed in Section 8.16 arise. The drawback to the ChimeraVP approach is that each configuration holds such a small subset of the overall hypertext; the composite approach holds larger subsets, and hence provides better structure versioning.

Chapter 10

Related Work

In the process of presenting a domain model for hypertext versioning, this document has performed an extensive analysis of the hypertext versioning literature, often bringing in related work from the neighboring disciplines of hypertext, configuration management, computer supported cooperative work, and document management. However, there are some pieces of related work whose scope of inquiry is either a whole, or significant portion of, a domain. Often these are domains other than hypertext versioning, though there has been some limited work in this domain as well. This related work can take the form of a survey paper, wherein a significant aspect of a domain is analyzed in the process of the survey, but typically not having the same structural organization into domain characterization and domain model as the domain analysis presented within. Alternately, there are examples of the analysis of full domains that come from the software reuse community, and hence do share organizational similarities to this work.

Within the hypertext community, the Dexter hypertext reference model effort can be viewed as an example of domain analysis for hypertext systems [94]. The Dexter group included researchers involved in the creation of many early hypertext systems, including NLS/Augment, Concordia/Document Examiner, HyperCard, Hyperties, IGD, Intermedia, KMS, Neptune/HAM, NoteCards, the Sun Link Service, and TEXTNET [95]. Together, they developed a common set of terms to describe hypertext abstractions, such as anchor, link, atom, and composite. Some requirements common to hypertext systems were captured. Though the Dexter group was unaware of the domain analysis work being performed in the software reuse community, the common terms and requirements, along with the set of systems that participated, can be viewed as a characterization of the hypertext systems domain circa 1988-1990. The Dexter group also produced a high level architectural model of hypertext systems, breaking them down into a run-time layer, a storage layer, and a within-component layer. This was augmented by specific operations that could be performed at each layer, and these operations were formally specified using the Z language in the workshop's final report [94]. The operations and their formal specification can be viewed as a domain model for hypertext systems. After the Dexter group had concluded, the DHM system implemented the Dexter architecture, and refined its notion of composite [85,84]. This work can be viewed as implementing the domain reference architecture within the Dexter model.

Within hypertext versioning Fabio Vitali has written the only published survey, a short paper that belongs to a special issue of ACM Computing Surveys on hypertext [193]. This paper provides a good overview of hypertext versioning, presenting its advantages for history recording, work accountability, collaboration, and reference permanence. The paper also presents a brief overview of some hypertext versioning issues, and a brief history of hypertext versioning. Despite being a good introduction to the hypertext versioning literature, this paper does not contain sufficient detail to be considered a full domain analysis. Of course, that was not its objective.

The Hypermedia Version Control Framework by Hicks et al. presents the HURL data model, and a conceptual architecture [100]. The HURL data model extends the SP3/HB3 [116] data model by making every object versionable. The conceptual architecture adds a version control component to an open hyperbase system [142], wherein applications that have been integrated to work with the system can interact with a set of open interfaces to perform hypertext capability. Though the HURL data model builds upon the experiences of other hypertext versioning research, especially HyperPro [141] and CoVer [87], its strong roots in the SP3/HB3 data model, instead of an analysis of existing hypertext versioning systems, make it a weak starting point for a domain model. Though a brief description of mapping the CoVer data model into HURL was provided in [100], much work would need to be performed to model the complex containment structures of the DeltaV data model, or the revision selection behavior of HyperPro or HyperProp [179]. Furthermore, the act of mapping these data models into HURL would cause them to lose their nuances and subtleties in favor of the normative data model. The approach taken in this dissertation of representing the data models using containment relationships allows each data model to be expressed with less loss of detail. However, the Hypermedia Version Control Framework is still the first effort to comprehensively describe the data modeling issues inherent to hypertext versioning, and thus it can be viewed as performing a preliminary domain analysis. Unlike this work, which is only an analysis, the Hypermedia Version Control Framework was implemented by extending the HB3/SP3 system, providing a concrete instance of the conceptual architecture and domain model.

Similar to the Hypermedia Version Control Framework, the HyperForm system provides a hyperbase containing a version control component, and demonstrated models of the hypertext capabilities of Neptune/HAM and the Danish HyperBase [202]. The later HyperDisco system extended this work [204]. This work focuses primarily on the architecture needed to flexibly represent and implement various hypertext data models, including components for concurrency control, notification control, version control, access control, along with query and search. The drawback to this work is its lack of flexibility. The concurrency control component only supports locking and short database transactions, while the version control component supports only state-based versioning. Thus these components are not be able to model the full range of capabilities found in the hypertext versioning domain. Nevertheless, this architecture, along with the one in the Hypermedia Version Control Framework, provides a good starting point for the development of a domain reference architecture.

Outside of hypertext versioning, two survey articles are relevant to hypertext versioning: the Conradi and Westfechtel survey of versioning data models [35] in versioning and configuration management systems, and the Katz survey of versioning in engineering database systems [108]. Not surprisingly, by focusing on a different domain, Conradi and Westfechtel found different data models. In addition to the predominantly state-based versioning, where each revision has distinct, persistent identity, which Conradi and Westfechtel term *extensional* versioning, they also discuss *intensional* versioning, where revisions are constructed from property-based descriptions. They also provide a more detailed discussion of change-based versioning, a taxonomy of versioning data models, as well as a discussion of the interplay of "product" space and version space. Due to its broad consideration of versioning and configuration management systems, this work will be integrated with the data model herein as the domain of inquiry is expanded from hypertext versioning out to include versioning and configuration management systems.

Many of the versioning issues encountered in hypertext versioning and configuration management are also found in engineering database systems that support the development of integrated circuits. Randy Katz performed a substantive survey of the version data models in engineering database systems [108]. Similar to this work, Katz's survey used as basic modeling primitives derivation (predecessor/successor), composition (containment), and variant (is-kind-of) relationships. These were then used to show how engineering database systems model various versioning and work scenarios. The survey provides a set of unified terminology, a unified data model, and a high-level conceptual architecture, and thus has many of the outputs associated with domain analysis. Due to the similarity in modeling approach, this work will be important when merging engineering database systems into this domain analysis.

Much has been written on the general practice of domain analysis. Descriptions and processes for performing domain analysis are provided by Arango in [9] and his dissertation [10], by Prieto-Díaz in [155,156], and by Hooper and Chester in [102], p. 51-66. The domain-specific software architecture (DSSA) approach is described by Tracz in [187,188], by Might in [131], and in curriculum form by Taylor, Tracz, and Coglianese in [184].

Chapter 11

Future Work

As a domain, hypertext versioning combines a wide range of version control, collaboration, hypertext, and user interface issues, with a relatively small set of systems that have explored the design space so far. As a result, it is not at all surprising that the systematic organization of hypertext versioning knowledge performed during this domain analysis results in a number of broad, and interesting avenues for future research.

When we first defined the hypertext versioning domain, the definition only included those systems that whose links could be traversed using a hypertext style of navigation. However, since links are very similar to the kinds of relationships that also exist in document management, engineering databases [108], and configuration management systems [35], it is a natural direction to expand the domain description to fully encompass these related domains. Indeed, recent work within the SCM community by Estublier et al. on similarities with product data management (PDM) [62], and by Westfechtel and Conradi on parallels with engineering data management (EDM) [197], highlight a growing awareness of this domain overlap. Where possible, aspects from these other domains have been included in this document, but only the hypertext versioning domain was fully explored and represented. Future work will explore these other domains in depth, and will expand the model based on new abstractions, terminology, requirements, and design spaces gathered from this examination. Ideally this work will be the foundation for a new domain, that of *support environments for complex information artifacts*.

This domain analysis has not had code reuse as goal, choosing instead to develop a characterization of the domain, and a domain model. In a typical domain analysis, these steps are followed by the creation of a reference architecture for the domain, and reusable components can be developed for this architecture, sometimes using automatic code generation [16,17,136,10]. Especially since the design spaces for containment and representation of revisions have been well parameterized in this model, it would be a straightforward extension to create a reference architecture, and investigate automatic generation of components from a formal specification. We envision a "YACC (Yet Another Compiler-Compiler) for hypertext versioning" where the primary input is a formal description of the data model of the system, and the output is a repository that implements the data model. We expect that the formal model will need to be augmented, as YACC files are, with a description of the semantics associated with elements of the data model. The system would also have some built-in functions to handle common operations, such as link traversal, or compound documents.

The representation of variants within hypertext versioning systems is a relatively unexplored area, since only the CoVer [87] system has addressed this issue at all. Configuration management and document management systems have tried several approaches that have not yet been used for hypertext versioning, such as within-object variants, change-based approaches, and system models. Exploring these alternate techniques, along with a thorough exploration of variant management for hypertext systems in general, would be a valuable addition to this work.

Composite-based hypertext versioning systems tend to use branching as their concurrency control technique. However, there are several concurrency control techniques that do not use branching. In particular, it would be interesting to employ a synchronous editor that uses operational transforms [58] in a hypertext versioning system. Such a system would not need to use branching for version control, and this might make it easier to represent other kinds of variants using branching. In general, the insight that variant management and concurrency control are entangled issues should be further explored to tease out the truly orthogonal aspects of the combined variant and concurrency control design space.

To date, user interface issues have not been the focus of investigation in hypertext versioning systems. This observation also holds for configuration management systems too. Research that focuses on data modeling and system modeling has been more the norm. Since there have been concerns raised about the user-perceived complexity of hypertext versioning systems [141], exploration of user interfaces for hypertext versioning systems, and for containment structures in general would provide further information on whether these systems are too complex. Visualizing containment data structures for system builders is also an issue. If it is possible to automatically generate a hypertext versioning repository based on a formal model, is it also possible to automatically generate the containment data model diagrams? The realm of possible containment structures is vast, and the ability for humans to understand them is limited. Automatic visualization of containment structures could help this understanding process. Another user interface issue is how to visualize and control versioned link traversal at the user interface. Users ideally would like to know the revision and time of the object they are about to navigate into.

Providing awareness of other collaborator's state and activities is an issue in many collaborative systems, and hypertext versioning is no exception. However, this area has not been thoroughly investigated. Due to the influence of time in the system, hypertext versioning adds new requirements to the general awareness problem. For example, it would be nice to know what time another user is navigating at (e.g., what is the current setting on the time dial of their way-back machine?) Also, knowing the revision of each object and container that is being worked on would also aid awareness. It would also be useful to know where in the whole hypertext structure other collaborators are working.

An important issue for composite-based systems is controlling and visualizing hypertext network merges, since this is how collaborators combine their separate strands of work. Yet, this area has received very little research attention. Since the structural aspects of merging depend on the system's data structure, it might be possible to automatically create a merge tool from a description of the system's data model. One area that began to be explored in [90] was visualizations support the merge process. However, these user interfaces were never tested against actual users. We would like to know which visualizations users find it easy to understand, and which are difficult? Importantly, is the concept of merging hypertext networks too confusing for general computer users?

Finally, this work has almost entirely focused on versioning of static objects and links. However, dynamic content abounds on the Web, and research systems such as Microcosm [43] have highlighted the utility of dynamic links. The general problem of versioning of dynamic content and dynamic links is unsolvable, since too many controlling variables (e.g., hardware characteristics, operating system version) are outside the realm of control of the hypertext versioning system. However, there might be some middle ground where the hypertext versioning system could work in tandem with an application in the user's environment to control some system aspects normally considered out of bounds, and thereby provide some versioning capability for dynamic objects and links. Such an approach might work in a trusted environment; security would certainly be an important issue in such a system.

Chapter 12

Contributions and Conclusion

Taken as a whole, the major contribution of this work is its systematic organization of the vast majority of information concerning hypertext versioning. This was accomplished by performing a domain analysis over existing research in hypertext versioning, at times drawing upon relevant research in the disciplines of configuration management, computer-supported cooperative work, and document management. The field of domain analysis, traditionally used to foster software reuse, was employed here to take advantage of its framework for organizing information about a specific domain. This is the first application of domain analysis techniques on hypertext systems. Furthermore, by concentrating solely on reuse of the domain characterization and domain model, and not on actual code reuse, this is an unusual application of domain analysis techniques.

The hypertext versioning domain analysis yielded a characterization of the domain, consisting of domain terminology, a taxonomy of hypertext versioning systems, and a set of domain reference requirements. All of these outputs of domain characterization are novel contributions. By detailing the origin of the terms anchor and domain, and by cataloging the terms systems use for their linkable information and for anchors, the domain terminology has increased our knowledge of these terms. Prior to this work it was not known that Norm Meyrowitz coined the term anchor in the mid to late 1980's, or that the term domain has roots back into Artificial Intelligence. The taxonomy of hypertext versioning systems organized systems into five categories: versioning for reference permanence, systems that version data, but not structure, composite-based, Web versioning, and open hypertext versioning. This categorization of systems is a first for hypertext versioning, and encompasses work that has been published in both the hypertext, and configuration management communities on hypertext and Web versioning. The comprehensive collection of reference requirements is also a solid contribution, capturing more requirements, in greater detail, than previous summaries such as [154] and [179] which provided a foundation for this work.

The second major output of domain analysis, the domain model, itself contains many significant contributions. The first aspect of the domain model is the data modeling model. By focusing on containment relationships within hypertext systems, this data modeling model has exposed both that containment is inherent to all hypertext systems, and that data models cannot be fully understood without also understanding the containment relationships. Most existing hypertext systems provide only an object oriented inheritance hierarchy to describe their data models. However, the experience of creating a standard collaborative work scenario across several composite-based systems showed that inheritance relationships offer little explanatory power when describing how the system represents hyperdocuments and working areas, while containment relationships, by highlighting how container objects and links hold other objects, provided a much better understanding. Finally, the act of using a consistent approach to modeling the data models of several hypertext, and hypertext versioning systems, makes the similarities and differences among their data models more consistent.

Modeling static hypertext links as containers, using the same set of containment relationships, is another contribution. Viewing links as containers allows the data models of hypertext versioning and configuration management systems to both be characterized as systems of containment relationships. While this robs links of some of their special status, it allows these different classes of systems to be examined in a more commensurable way, teasing out similarities previously hidden by terminology.

This examination of the containment properties of hypertext versioning systems would not have been possible without a solid model of containment. While containment has often been employed and discussed, very few researchers have moved beyond the deceptive categories of reference and inclusion containment to characterize all of the subtleties in containment. This comprehensive model of containment has applicability to a wide range of object management and file systems.

The second aspect of the domain model, the characterization of the design spaces for meeting the domain requirements, also resulted in several contributions. The very expression of this information as a space of design choices is a useful contribution, since existing work tends to explore a subset of the entire space. By describing all choices, along with their tradeoffs, a more holistic view of each area emerges. Tying these design spaces explicitly to specific requirements also exposes cases where little is known, and more research needs to be performed.

The three-layer model of the state-based versioning design space provides a clean separation of concerns between the abstract notion of a revision history, the three major approaches for representing that history (versioned object, within-object, and predecessor/successor relationships), and the many ways these approaches can be concretely realized. This allows the characteristics of each approach to be modeled and evaluated independent of their specific concrete realization, and bases the variation among versioning approaches on differences in containment relationships between the versioned object, revisions, and predecessor/successor relationships. The description of the link versioning design space as an application of the general versioning design space is novel, since existing hypertext versioning systems have only provided a single link versioning mechanism, without exploring design tradeoffs.

Building on the model of containment, and the versioning design space, the structure versioning design space concisely describes a range of techniques for recording the history of hypertext structures. The major aspects of this space are a determination of the objects contained by the structure container, the versioning design space choice for the structure container and its containees, the containment design space choice for all container/containee pairs, and the location and scope of revision selection rules. Where previously composite-based hypertext versioning systems have explored only individual points, the structure versioning design space teases out their commonality, providing a map of design possibilities and a coherent model for describing the structure versioning capabilities of existing systems.

An unexpected result from this work is the recognition that the concept of variance encompasses both the notion of revisions, which are time-based variants, and concurrency control, concerned with per-user variants. The taxonomy of concurrency control techniques in terms of the lifespan of per-user variants is useful because it spans the branch-based concurrency control employed in versioning systems, as well as other concurrency control techniques that assume no versioning is in use.

Finally, though readers holding this document may find it hard to contemplate, entering into this project it was unknown whether domain analysis techniques could usefully be applied to hypertext versioning systems, since this domain is more abstract, and disparate than many prior foci of domain analysis. The standard outputs of domain analysis, a characterization of the domain, and a domain model, have in practice provided a useful structure for separating concerns within the hypertext versioning domain.

This work's primary motivation is to enhance the state of hypertext versioning knowledge so that future engineers creating systems for use in software engineering, document management, audits, law, and archives, would be able to quickly learn what is known about hypertext versioning. This knowledge would allow them to add hypertext capabilities to systems that otherwise would not, due to the lack of understanding concerning the interactions of links and the versioned objects that populate these use cases. However, whether this work will, in fact, have this effect is still unknown. To achieve this goal, the results in this document need to be distributed. In this respect, things are off to an inauspicious start, as dissertations are notorious for their lack of readership. Additionally, further work needs to be done to embrace different types of object management systems, not just those encountered in hypertext systems. The more broadly this work applies, the more likely a particular project will find it relevant. Though a significant work in its own right, this dissertation is just a way station. Much work remains to be done.

References

[1] M. Abu-Shakra and G. L. Fisher, “Multi-Grain Version Control in the Historian System,” *Proc. System Configuration Management Symposium (SCM-8)*, Brussels, Belgium, July 20-21, 1998, pp. 46-56.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, Mass.: Addison-Wesley, 1983.

[3] R. Akscyn, D. McCracken, and E. Yoder, “KMS: A Distributed Hypermedia System for Managing Knowledge In Organizations,” *Proc. ACM Hypertext'87 Workshop*, Chapel Hill, NC, Nov. 13-15, 1987, pp. 1-20.

[4] R. M. Akscyn, D. L. McCracken, and E. A. Yoder, “KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations,” *Communications of the ACM*, vol. 31, no. 7 (1988), pp. 820-835.

[5] J. Amsden, C. Kaler, and J. Stracke, “Goals for Web Versioning,” IBM, Microsoft, Netscape. Internet-Draft, work-in-progress, *draft-ietf-webdav-version-goals-01*, June 26, 1999.

[6] K. M. Anderson, “Integrating Open Hypermedia Systems with the World Wide Web,” *Proc. Eighth ACM Conference on Hypertext (Hypertext'97)*, Southampton, UK, April 6-11, 1997, pp. 157-166.

[7] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr., “Chimera: Hypertext for Heterogeneous Software Environments,” *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 94-107.

[8] Apple Computer, *HyperCard Script Language Guide*. Reading, MA: Addison-Wesley, 1988.

[9] G. Arango, “Domain Analysis Methods,” in *Software Reusability*, W. Schäfer, R. Prieto-Díaz, and M. Matsumoto, Eds. New York: Ellis Horwood, 1994, pp. 17-49.

[10] G. F. Arango, “Domain Engineering for Software Reuse,” Ph.D. Dissertation. University of California, Irvine, Irvine, CA, 1988.

[11] D. L. Atkins, “Version Sensitive Editing: Change History as a Programming Tool,” *Proc. System Configuration Management Symposium (SCM-8)*, Brussels, Belgium, July 20-21, 1998, pp. 146-157.

[12] A. Babich, J. Davis, R. Henderson, D. Lowry, S. Reddy, and S. Reddy, “DAV Searching and Locating,” FileNet, CourseNet, Netscape, Novell, Microsoft, Oracle. Unpublished manuscript, intended for submission as an Internet-Draft, *draft-davis-dasl-protocol-00*, April 20, 2000.

[13] T. A. Ball and S. G. Eick, “Software visualization in the large,” *IEEE Computer*, vol. 29, no. 4 (1996), pp. 33-43.

[14] R. M. Balzer, “A Global View of Automatic Programming,” *Proc. Third Joint Conference on Artificial Intelligence*, SRI International, August, 1973, pp. 494-499.

[15] A. Bapat, J. Wäsch, K. Aberer, and J. M. Haake, “HyperStorM: An Extensible Object-Oriented Hypermedia Engine,” *Proc. Seventh ACM Conference on Hypertext (Hypertext '96)*, Washington, DC, March 16-20, 1996, pp. 203-214.

[16] D. Batory, L. Coglianese, S. Shafer, and W. Tracz, “The ADAGE Avionics Reference Architecture,” University of Texas, Austin, Texas, Technical Report ADAGE-UT-94-03, 1994.

[17] D. Batory and Y. Smaragdakis, “Another Look at Architectural Styles and ADAGE,” University of Texas, Austin, Texas, Technical Report UT-ADAGE-95-02, 1995.

[18] L. Bendix and F. Vitali, “VTML for Fine-Grained Change Tracking in Editing Structured Documents,” *Proc. 9th Int'l Symposium on System Configuration Management (SCM-9)*, Toulouse, France, Sept. 5-7, 1999, pp. 139-156.

[19] R. Bentley, T. Horstmann, and J. Trevor, “The World Wide Web as Enabling Technology for CSCW: The Case of BSCW,” *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 6, no. 2-3 (1997), pp. 111-134.

[20] B. Berliner, “CVS II: Parallelizing Software Development,” *Proc. Winter 1990 USENIX Conference*, Washington, DC, Jan. 22-26, 1990, pp. 341-351.

[21] T. Berners-Lee, “Versioning", a web page that is part of the original design notes for the WWW, (1990). Web page, accessed Nov. 15, 1999. http://web1.w3.org/DesignIssues/Versioning.html.

[22] T. Berners-Lee, R. Fielding, and L. Masinter, “Uniform Resource Identifiers (URI): Generic Syntax,” MIT/LCS, U.C. Irvine, Xerox Corporation. Internet Draft Standard Request for Comments (RFC) 2396, August, 1998.

[23] G. Booch, *Object Oriented Design with Applications*. Redwood City: Benjamin/Cummings, 1991.

[24] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, “Extensible Markup Language (XML),” World Wide Web Consortium Recommendation REC-xml. .

[25] V. Bush, “As We May Think,” *Atlantic Monthly*, July, 1945, pp. 101-108.

[26] CAMP, “Common Ada Missile Packages, Final Technical Report, Vols. 1, 2, and 3,” Air Force Armament Laboratory, Eglin AFB, FL AD-B-102 654, 655, 656, 1987.

[27] M. A. Casanova, L. Tucherman, M. J. D. Lima, J. L. R. Netto, N. Rodriguez, and L. F. G. Soares, “The Nested Composite Model for Hyperdocuments,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, TX, Dec. 15-18, 1991, pp. 193-202.

[28] L. M. Chan, *Immroth's Guide to the Library of Congress Classification*, Third ed. Littleton, CO: Libraries Unlimited, Inc., 1980.

[29] P. Chen, “The Entity-Relationship Model: Toward a Unified View of Data,” *ACM Trans. on Database Systems*, vol. 1, no. 1 (1976), pp. 9-36.

[30] Y.-F. Chen, F. Douglis, H. Huang, and K.-P. Vo, “TopBlend: An Efficient Implementation of HtmlDiff in Java,” AT&T Labs, Florham Park, NJ, Technical Report TR 00.5.1, 2000.

[31] G. Clemm, J. Amsden, C. Kaler, and J. Whitehead, “Versioning Extensions to WebDAV,” Rational, IBM, Microsoft, U.C. Irvine. Internet-Draft, work-in-progress, *draft-ietf-deltav-versioning-05*, June 19, 2000.

[32] M. Colton, “Preserving the Web's Digital History,” *Brill's Content*, November, 1999, pp. 54-56.

[33] J. Conklin, “Hypertext: An Introduction and Survey,” *IEEE Computer*, vol. 20, no. 9 (1987), pp. 17-41.

[34] J. Conklin and M. L. Begeman, “gIBIS: A Hypertext Tool for Team Design Deliberation,” *Proc. ACM Hypertext'87 Workshop*, Chapel Hill, NC, Nov. 13-15, 1987, pp. 247 - 251.

[35] R. Conradi and B. Westfechtel, “Version Models for Software Configuration Management,” *ACM Computing Surveys*, vol. 30, no. 2 (1998), pp. 232-282.

[36] Continuus Software, “Continuus/CM Product Description,” (1999). Web page, accessed Nov., 1999, http://www.continuus.com/products/productsBB.html.

[37] J. O. Coplien, D. L. DeBruler, and M. B. Thompson, “The Delta System: A Nontraditional Approach to Software Version Management,” *Proc. International Switching Symposium*, Phoenix, Arizona, March 15, 1987, 1987, pp. 181-197.

[38] M. L. Creech, D. F. Freeze, and M. L. Griss, “Using Hypertext In Selecting Reusable Software Components,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 25-38.

[39] S. Dart, “Concepts in Configuration Management Systems,” *Proc. Third Int'l Workshop on Software Configuration Management*, Trondheim, Norway, June 12-14, 1991, pp. 1-18.

[40] C. Darwin, *The Origin of Species by Means of Natural Selection*, Sixth ed. New York: 1873 Press, 1872.

[41] H. C. Davis, “Data Integrity Problems in an Open Hypermedia Link Service,” Dissertation. University of Southampton, Southampton, UK, 1995.

[42] H. C. Davis, “Referential Integrity of Links in Open Hypermedia Systems,” *Proc. Ninth ACM Conference on Hypertext (Hypertext'98)*, Pittsburgh, PA, June 20-24, 1998, pp. 207-216.

[43] H. C. Davis, W. Hall, I. Heath, G. Hill, and R. Wilkins, “Towards an Integrated Information Environment with Open Hypermedia Systems,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 181-190.

[44] H. C. Davis, S. Knight, and W. Hall, “Light Hypermedia Link Services: A Study of Third Party Application Integration,” *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 41-50.

[45] N. Delisle and M. Schwartz, “Neptune: A Hypertext System for CAD Applications,” *Proc. Int'l Conference on the Management of Data (SIGMOD'86)*, Washington, DC, May 28-30, 1986, pp. 132-143.

[46] N. M. Delisle and M. D. Schwartz, “Contexts-A Partitioning Concept for Hypertext,” *ACM Transactions on Office Information Systems*, vol. 5, no. 2 (1987), pp. 168-186.

[47] S. DeRose, E. Maler, D. Orchard, and B. Trafford, “XML Linking Language (XLink),” World Wide Web Consortium Candidate Recommendation CR-xlink-20000703, July 3, 2000.

[48] S. J. DeRose, “Expanding the Notion of Links,” *Proc. Hypertext'89*, Pittsburgh, PA, Nov. 5-8, 1989, pp. 249-258.

[49] L. DeYoung, “Hypertext Challenges in the Auditing Domain,” *Proc. Hypertext'89*, Pittsburgh, PA, Nov. 5-8, 1989, pp. 169-180.

[50] DMA Technical Committee, *DMA 1.0 Specification*. Silver Spring, Maryland: AIIM International, 1998.

[51] F. Douglis, T. Ball, Y.-F. Chen, and E. Koutsofios, “The AT&T Internet Difference Engine: Tracking and viewing changes on the Web,” *World Wide Web*, vol. 1, no. 1 (1998), pp. 27-44.

[52] P. Dourish, “The Parting of the Ways: Divergence, Data Management and Collaborative Work,” *Proc. Fourth European Conference on Computer Supported Cooperative Work (ECSCW'95)*, Stockholm, Sweden, Sept. 10-14, 1995, pp. 215-230.

[53] P. Dourish and V. Bellotti, “Awareness and Coordination in Shared Workspaces,” *Proc. Computer Supported Cooperative Work (CSCW'92)*, Toronto, Canada, Oct. 31-Nov. 4, 1992, pp. 107-114.

[54] S. Dreilinger, “CVS Version Control for Web Site Projects,” (1998). Web page, accessed Nov. 22, 1999. http://interactive.com/cvswebsites/.

[55] D. Durand and P. Kahn, “MAPA: a system for inducing and visualizing hierarchy in websites,” *Proc. Ninth ACM Conference on Hypertext and Hypermedia (Hypertext'98)*, Pittsburgh, PA, June 20-24, 1998, pp. 66-76.

[56] D. G. Durand, “Palimpsest: Change-Oriented Concurrency Control for the Support of Collaborative Applications,” Ph.D. Dissertation. Boston University, Boston, MA, 1999.

[57] W. K. Edwards and E. D. Mynatt, “Timewarp: Techniques for Autonomous Collaboration,” *Proc. ACM Conference on Human Factors in Computing Systems (CHI'97)*, Atlanta, GA, March 22-27, 1997, pp. 218 - 225.

[58] C. A. Ellis and S. J. Gibbs, “Concurrency Control in Groupware Systems,” *Proc. ACM SIGMOD'89 Conference on the Management of Data*, Seattle, WA, May 2-4, 1989.

[59] D. C. Engelbart, “Letter to Vannevar Bush and Program On Human Effectiveness,” in *From Memex to Hypertext: Vannevar Bush and the Mind's Machine*, J. M. Nyce and P. Kahn, Eds. Boston: Academic Press, 1991, pp. 235-236.

[60] D. C. Engelbart and W. K. English, “A Research Center for Augmenting Human Intellect,” *Proc. 1968 Fall Joint Computer Conference (AFIPS)*, San Francisco, CA, December, 1968, 1968, pp. 395-410.

[61] J. Estublier and R. Casallas, “The Adele Configuration Manager,” in *Configuration Management*, W. F. Tichy, Ed. Chicester: John Wiley & Sons, 1994, pp. 99-133.

[62] J. Estublier, J.-M. Favre, and P. Morat, “Toward SCM/PDM Integration,” *Proc. System Configuration Management (SCM-8)*, Brussels, Belgium, July 20-21, 1998, pp. 75-94.

[63] European Computer Manufacturers Association (ECMA), “ECMAScript Language Specification, 3rd Edition,” Standard ECMA-262, December 1999.

[64] European Computer Manufacturers Association (ECMA) and National Institute of Standards and Technology (NIST), “Reference Model for Frameworks of Software Engineering Environments,” ECMA/NIST ECMA TR/55, NIST Special Publication 500-211, 1993.

[65] P. H. Feiler, “Configuration Management Models in Commercial Environments,” Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU/SEI-91-TR-7, March 1991.

[66] J. Feise, “Accessing the History of the Web: A Way-Back Machine,” *Proc. Sixth Workshop on Open Hypermedia Systems (OHS-6)*, San Antonio, TX, May 30, 2000.

[67] J. C. Ferrans, D. W. Hurst, M. A. Sennett, B. M. Covnot, W. Ji, P. Kajka, and W. Ouyang, “HyperWeb: A Framework for Hypermedia-Based Environments,” *Proc. ACM SIGSOFT'92: Fifth Symposium on Software Development Environments*, Washington, DC, December, 1992, pp. 1-10.

[68] R. Fielding, J. Gettys, J. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” UC Irvine, Compaq, W3C, Xerox, Microsoft. Internet Draft Standard Request for Comments (RFC) 2616, June, 1999.

[69] R. T. Fielding and R. N. Taylor, “Principled Design of the Modern Web Architecture,” *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 4-11, 2000, pp. 407-416.

[70] R. T. Fielding, E. J. Whitehead, Jr., K. M. Anderson, G. A. Bolcer, P. Oreizy, and R. N. Taylor, “Web-Based Development of Complex Information Products,” *Communications of the ACM*, vol. 41, no. 8 (1998), pp. 84-92.

[71] M. Foucault, *The Archaeology of Knowledge*. New York: Pantheon Books, 1972.

[72] A. M. Fountain, W. Hall, I. Heath, and H. C. Davis, “Microcosm: An Open Model for Hypermedia with Dynamic Linking,” *Proc. First European Conference on Hypertext (ECHT'90)*, Versailles, France, Nov. 27-30, 1990, pp. 298-311.

[73] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication,” Northwestern University, Verisign, AbiSource, Agranat Systems, Microsoft, Netscape Communications, Open Market. Internet Draft Standard Request for Comments (RFC) 2617, June, 1999.

[74] C. W. Fraser and E. W. Myers, “An Editor for Revision Control,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 2 (1987), pp. 277-295.

[75] P. Fröhlich and W. Nejdl, “WebRC: Configuration Management for a Cooperation Tool,” *Proc. 7th Workshop on Software Configuration Management (SCM-7)*, Boston, MA, May 18-19, 1997, pp. 175-185.

[76] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[77] P. K. Garg, “Abstraction Mechanisms in Hypertext,” *Communications of the ACM*, vol. 31, no. 7 (1988), pp. 862-870.

[78] P. K. Garg and W. Scacchi, “A Software Hypertext Environment for Configured Software Descriptions,” *Proc. International Workshop on Software Version and Configuration Control*, Grassau, Germany, January, 1988, pp. 326-343.

[79] Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen, "HTTP Extensions for Distributed Authoring -- WEBDAV," Microsoft, U.C. Irvine, Netscape, Novell. Internet Proposed Standard Request for Comments (RFC) 2518, Feburary, 1999.

[80] I. P. Goldstein and D. P. Bobrow, "A Layered Approach to Software Design," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York, NY: McGraw-Hill, 1984, pp. 387-413.

[81] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Reading, MA: Addison-Wesley, 1996.

[82] A. Grafton, *The Footnote: A Curious History*. Cambridge, MA: Harvard University Press, 1997.

[83] M. Griss, "Implementing Product-Line Features with Component Reuse," *Proc. 6th International Conference on Software Reuse*, Vienna, Austria, June, 2000.

[84] K. Grønbæk, "Composites in a Dexter-Based Hypermedia Framework," *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 59-69.

[85] K. Grønbæk and R. H. Trigg, "Design Issues for a Dexter-based hypermedia system," *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 191-200.

[86] K. Grønbæk and R. H. Trigg, *From Web to Workplace: Designing Open Hypermedia Systems*. Cambridge, MA: MIT Press, 1999.

[87] A. Haake, "CoVer: A Contextual Version Server for Hypertext Applications," *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 43-52.

[88] A. Haake, "Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications," *Proc. Sixth ACM Conference on Hypertext (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 81-93.

[89] A. Haake and J. Haake, "Take CoVer: Exploiting Version Support in Cooperative Systems," *Proc. InterCHI'93 - Human Factors in Computer Systems*, Amsterdam, Netherlands, April, 1993, pp. 406-413.

[90] A. Haake, J. Haake, and D. Hicks, "On Merging Hypertext Networks," *Proc. Workshop on the Role of Version Control in CSCW*, 1995.

[91] A. Haake and D. Hicks, "VerSE: Towards Hypertext Versioning Styles," *Proc. Seventh ACM Conference on Hypertext (Hypertext '96)*, Washington, DC, March 16-20, 1996, pp. 224-234.

[92] J. Haake, C. M. Neuwirth, and N. A. Streitz, "Coexistence and Transformation of Informal and Formal Structures: Requirements for More Flexible Hypermedia Systems," *Proc. 1994 European Conference on Hypermedia Technology (ECHT'94)*, Edinburgh, Scotland, Sept. 18-23, 1994, pp. 1-12.

[93] J. M. Haake and B. Wilson, "Supporting Collaborative Writing of Hyperdocuments in SEPIA," *Proc. Computer Supported Cooperative Work (CSCW'92)*, Toronto, Canada, Oct. 31-Nov. 4, 1992, pp. 138-146.

[94] F. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," *Proc. NIST Hypertext Standardization Workshop*, Gaithersburgh, MD, Jan 16-18, 1990, pp. 95-133.

[95] F. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," *Communications of the ACM*, vol. 37, no. 2 (1994), pp. 30-39.

[96] F. G. Halasz, "Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems," *Communications of the ACM*, vol. 31, no. 7 (1988), pp. 836-852.

[97] F. G. Halasz, ""Seven Issues": Revisited, Hypertext'91 Closing Plenary," *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, TX, 1991.

[98] B. M. Hauzeur, "A Model for Naming, Addressing, and Routing," *ACM Transactions on Office Information Systems*, vol. 4, no. 4 (1986), pp. 293-311.

[99] F. Hayes-Roth, "Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program," . February, 1994.

[100] D. L. Hicks, J. J. Leggett, P. J. Nürnberg, and J. L. Schnase, "A Hypermedia Version Control Framework," *ACM Transactions on Information Systems*, vol. 16, no. 2 (1998), pp. 127-160.

[101] D. L. Hicks, J. J. Leggett, and J. L. Schnase, "Version Control in Hypertext Systems," Texas A&M University, Technical Report TAMU HRL-91-004, July 1991.

[102] J. W. Hooper and R. O. Chester, *Software Reuse: Guidelines and Methods*. New York: Plenum Press, 1991.

[103] R. Hull and R. King, “Semantic Database Modeling: Survey, Applications, and Research Issues,” *ACM Computing Surveys*, vol. 19, no. 3 (1987), pp. 201-260.

[104] J. J. Hunt, F. Lamers, J. Reuter, and W. F. Tichy, “Distributed Configuration Management via Java and the World Wide Web,” *Proc. 7th Workshop on Software Configuration Management (SCM-7)*, Boston, MA, May 18-19, 1997, pp. 161-174.

[105] Internet Archive, “Internet Archive Home Page,” (1999). Web page, accessed Dec. 5, 1999. http://www.archive.org/.

[106] C. J. Kacmar and J. J. Leggett, “PROXHY: A Process-Oriented Extensible Hypertext Architecture,” *ACM Trans. on Information Systems*, vol. 9, no. 4 (1991), pp. 399-419.

[107] F. Kappe and G. Pani, “Hyper-G Client-Server Protocol (HG-CSP),” in *Hyper-G, now, HyperWave*, H. Maurer, Ed. Harlow, England: Addison-Wesley, 1996, pp. 550-591.

[108] R. H. Katz, “Toward a Unified Framework for Version Modeling in Engineering Databases,” *ACM Computing Surveys*, vol. 22, no. 4 (1990), pp. 375-408.

[109] M. J. Knister and A. Prakash, “DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors,” *Proc. Computer-Supported Cooperative Work (CSCW'90)*, Los Angeles, CA, Oct. 7-10, 1990, pp. 343-355.

[110] H. Koike and H.-C. Chu, “How Does 3-D Visualization Work in Software Engineering? : Empirical Study of a 3-D Version/Module Visualization System,” *Proc. 1998 International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 19-25, 1998, pp. 516-519.

[111] V. Kruskal, “Managing Multi-Version Programs with an Editor,” *IBM Journal of Research and Development*, vol. 28, no. 1 (1984), pp. 74-81.

[112] S. Kydd, A. Dyke, and D. Jenkins, “Hypermedia Version Support for the Online Design Journal,” *Proc. Workshop on Versioning in Hypertext Systems*, Edinburgh, Scotland (held with ECHT'94), Sept. 18-23, 1994, pp. 9-12.

[113] S. Lawrence and C. L. Giles, “Accessibility and Distribution of Information on the Web,” *Nature*, vol. 400, (1999), pp. 107-109.

[114] J. Leggett, Email communication, May 3, 2000.

[115] J. J. Leggett, “Report of the Workshop on Hyperbase Systems,” Dept. of Computer Science, Texas A&M University, College Station, TX, Technical Report TAMU-HRL 93-009, 1993.

[116] J. J. Leggett and J. L. Schnase, “Viewing Dexter with Open Eyes,” *Communications of the ACM*, vol. 37, no. 2 (1994), pp. 76-86.

[117] J. J. Leggett, J. L. Schnase, J. B. Smith, and E. A. Fox, “Final Report of the NSF Workshop on Hyperbase Systems,” Texas A&M University, College Station, TX, Technical Report TAMU-HRL 93-002, 1993.

[118] E. Levinson, “The MIME Multipart/Related Content-type.” Internet Request for Comments (RFC) 2387, August 1998.

[119] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson, “Change Oriented Versioning in a Software Engineering Database,” *Proc. Second Internation Workshop on Software Configuration Management*, Princeton, NJ, October 24, 1989, pp. 56-65.

[120] C. Y. Lo, “Comparison of Two Approaches of Managing Links in Multiple Versions of Documents,” *Proc. Workshop on Versioning in Hypertext Systems*, Edinburgh, Scotland (held with ECHT'94), Sept. 18-23, 1994, pp. 17-22.

[121] J. MacDonald, P. N. Hilfinger, and L. Semenzato, “PRCS: The Project Revision Control System,” *Proc. System Configuration Mangement Symposium (SCM-8)*, Brussels, Belgium, July 20-21, 1998, pp. 33-45.

[122] T. W. Machan, “Chaucer's Poetry, Versioning, and Hypertext,” *Philological Quarterly*, vol. 73, no. 3 (1994), pp. 299-316.

[123] A. Mahler, “Variants: Keeping Things Together and Telling Them Apart,” in *Configuration Management*, W. F. Tichy, Ed. Chicester: John Wiley & Sons, 1994, pp. 73-97.

[124] C. Maioli, S. Sola, and F. Vitali, “Versioning for Distributed Hypertext Systems,” *Proc. Hypermedia'94*, Pretoria, South Africa, March, 1994.

[125] K. C. Malcolm, S. E. Poltrock, and D. Schuler, “Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 13-24.

[126] C. C. Marshall, F. G. Halasz, R. A. Rogers, and W. C. Janssen, Jr., “Aquanet: a hypertext tool to hold your knowledge in place,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 261-275.

[127] M. D. McIlroy, “Mass-Produced Software Components,” *Proc. 1968 NATO Conference on Software Engineering*, Garmisch, Germany, 1968, pp. 138-155.

[128] M. Melly and W. Hall, “Version Control in Microcosm,” *Proc. Workshop on the Role of Version Control in CSCW (held with ECSCW'95)*, Stockholm, Sweden, September, 1995.

[129] N. Meyrowitz, “Hypertext--Does It Reduce Cholesterol, Too?,” in *From Memex to Hypertext: Vanevar Bush and the Mind's Machine*, J. M. Nyce and P. Kahn, Eds. Boston: Academic Press, 1989, pp. 287-318.

[130] N. Meyrowitz, Email communication. May 9, 2000.

[131] R. J. Might, “Domain Models, What are they? How are they used?,” (1994?). Unpublished manuscript, accessed May 16, 2000. http://www.lmowego.com/owegofs/dssa/Domain_Models_Mite.ps.

[132] D. B. Miller, R. G. Stockton, and C. W. Krueger, “An Inverted Approach to Configuration Management,” *Proc. Second International Workshop on Software Configuration Management (SCM-2)*, Princeton, NJ, October 24, 1989, pp. 1-4.

[133] P. Mockapetris, “Domain Names - Concepts and Facilities,” ISI. Internet Request for Comments (RFC) 1034, November, 1987.

[134] Mortice Kern Systems, “Web Integrity,” (1999). Web page, http://www.mks.com/solution/wi/.

[135] J. Nanard and M. Nanard, “Using Structured Types to Incorporate Knowledge in Hypertext,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 329-343.

[136] J. M. Neighbors, “Software Construction Using Components,” Ph.D. Dissertation. Univ. of California, Irvine, Irvine, CA, 1980.

[137] T. H. Nelson, *Literary Machines*, 93.1 ed. Sausalito, CA: Mindful Press, 1981.

[138] T. H. Nelson, “As We Will Think,” in *From Memex to Hypertext: Vannevar Bush and the Mind's Machine*, J. M. Nyce and P. Kahn, Eds. Boston: Academic Press, 1991, pp. 245-260.

[139] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, “Computer Support for Distributed Collaborative Writing: Defining Parameters of Interaction,” *Proc. 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, Chapel Hill, NC, Oct. 22-26, 1994, pp. 145-152.

[140] J. Nielsen, “Hypertext'89 Trip Report,” (1989). Web page, accessed May 11, 2000. http://www.useit.com/papers/tripreports/ht89.html.

[141] K. Østerbye, “Structural and Cognitive Problems in Providing Version Control for Hypertext,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 33-42.

[142] K. Østerbye and U. K. Wiil, “The Flag Taxonomy of Open Hypermedia Systems,” *Proc. Seventh ACM Conference on Hypertext (Hypertext'96)*, Washington, DC, March 16-20, 1996, pp. 129-139.

[143] K. Ota, K. Takahashi, and K. Sekiya, “Version management with meta-level links via HTTP/1.1,” (1996). Internet-Draft (expired), accessed Nov., 1999, http://www.ics.uci.edu/pub/ietf/webdav/draft-ota-http-version-00.txt.

[144] F. Pacull, A. Sandoz, and A. Schiper, “Duplex: A Distributed Collaborative Editing Environment in Large Scale,” *Proc. 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, Chapel Hill, NC, Oct. 22-26, 1994, pp. 165-173.

[145] C. R. Palmer and G. V. Cormack, “Operation Transforms for a Distributed Shared Spreadsheet,” *Proc. Computer Supported Cooperative Work (CSCW'98)*, Seattle, WA, Nov. 14-18, 1998, pp. 69-78.

[146] D. L. Parnas, “On the Design and Development of Program Families,” *IEEE Trans. on Software Engineering*, vol. 2, no. 1 (1976), pp. 1-9.

[147] A. Pearl, “Sun's Link Service: A Protocol for Open Linking,” *Proc. Hypertext'89*, Pittsburgh, PA, Nov. 5-8, 1989, pp. 137-146.

[148] J. Peckham and F. Maryanski, “Semantic Data Models,” *ACM Computing Surveys*, vol. 20, no. 3 (1988), pp. 153-189.

[149] D. E. Perry, H. P. Siy, and L. G. Votta, “Parallel Changes in Large Scale Software Development: An Observational Case Study,” *Proc. 1998 International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 19-25, 1998, pp. 251-260.

[150] R. Pettengill and G. Arango, “Four Lessons Learned from Managing World Wide Web Digital Libraries,” *Proc. Second Annual Conference on the Theory and Practice of Digital Libraries*, Austin, TX, June 11-13, 1995.

[151] C. C. Pinter, *Set Theory*. Reading, Mass.: Addison-Wesley, 1971.

[152] J. E. Pitkow, “Summary of WWW Characterizations,” *Proc. 7th International World Wide Web Conference (WWW7)*, Brisbane, Queensland, Australia, April 14-18, 1998, pp. 551-558.

[153] Plato, *Dialogues -- Apology, Crito, Phaedo, Symposium, Republic*. New York: W. J. Black, 1942.

[154] V. Prevelakis, “Versioning Issues for Hypertext Systems,” in *Object Management*, D. Tsichritzis, Ed. Geneva, Switzerland: University of Geneva, 1990, pp. 89-106.

[155] R. Prieto-Díaz, “Domain Analysis for Reusability,” *Proc. Compsac'87*, Tokyo, Japan, Oct. 7-9, 1987, pp. 23-29.

[156] R. Prieto-Díaz, “Domain Analysis: An Introduction,” *ACM Software Engineering Notes*, vol. 15, no. 2 (1990), pp. 47-54.

[157] J. J. Putress and N. M. Guimaraes, “The Toolkit Approach to Hypermedia,” *Proc. European Conference on Hypertext (ECHT'90)*, Versailles, France, Nov. 27-30, 1990, pp. 25-37.

[158] Random House, *Webster's New Universal Unabridged Dictionary*. New York: Barnes & Noble, 1994.

[159] J. Raskin, “The Hype in Hypertext: A Critique,” *Proc. ACM Hypertext'87 Workshop*, Chapel Hill, NC, Nov. 13-15, 1987, pp. 325-330.

[160] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser, “An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors,” *Proc. Computer Supported Cooperative Work (CSCW'96)*, Boston, MA, Nov. 16-20, 1996, pp. 288-297.

[161] J. Reuter, S. U. Hänßgen, J. J. Hunt, and W. F. Tichy, “Distributed Revision Control Via the World Wide Web,” *Proc. Sixth Int'l Workshop on Software Configuration Management*, Berlin, Germany, March 25-26, 1996.

[162] A. Rizk and L. Sauter, “Multicard: An open hypermedia System,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 4-10.

[163] M. J. Rochkind, “The Source Code Control System,” *IEEE Transactions on Software Engineering*, vol. 1, no. 4 (1975), pp. 364-370.

[164] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[165] M. R. Salcedo and D. Decouchant, “Structured Cooperative Authoring for the World Wide Web,” *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 6, no. 2-3 (1997), pp. 157-174.

[166] N. Sarnak, R. Bernstein, and V. Kruskal, “Creation and Maintenance of Multiple Versions,” *Proc. International Workshop on Software Version and Configuration Control*, Grassau, Germany, 1988, pp. 264-275.

[167] M. A. Sasse, M. J. Handley, and S. C. Chuang, “Support for Collaborative Authoring via Email: The MESSIE Environment,” *Proc. Third European Conference on Computer-Supported Cooperative Work*, Milan, Italy, Sept. 13-17, 1993, pp. 249-264.

[168] J. L. Schnase, J. J. Leggett, D. L. Hicks, and R. L. Szabo, “Semantic Data Modeling of Hypermedia Associations,” *ACM Trans. on Information Systems*, vol. 11, no. 1 (1993), pp. 27-50.

[169] W. Schuler and J. B. Smith, “Author's Argumentation Assistant (AAA): A Hypertext-Based Authoring Tool for Argumentative Texts,” *Proc. First European Conference on Hypertext (ECHT'90)*, Versailles, France, Nov. 27-30, 1990, pp. 137-151.

[170] F. M. Shipman, III, J. Chaney, and G. A. Gorry, “Distributed Hypertext for Collaborative Research: The Virtual Notebook System,” *Proc. Hypertext '89*, Pittsburgh, PA, Nov. 5-8, 1989, pp. 129-135.

[171] F. M. Shipman, III, R. Furuta, D. Brenner, C.-C. Chung, and H.-w. Hsieh, “Using Path in the Classroom: Experiences and Adaptations,” *Proc. Ninth ACM Conference on Hypertext and Hypermedia (Hypertext'98)*, Pittsburgh, PA, June 20-24, 1998, 1998, pp. 267-276.

[172] B. Shneiderman, “User interface design for the Hyperties electronic encyclopedia,” *Proc. ACM Hypertext'87 Workshop*, Chapel Hill, NC, Nov. 13-15, 1987, pp. 189-194.

[173] J. Simonson, D. Berleant, X. Zhang, M. Xie, and H. Vo, “Version Augmented URIs for Reference Permanence via an Apache Module Design,” *Proc. WWW7, Computer Networks and ISDN Systems*, Brisbane, Australia, April 14-18, 1998, pp. 337-345.

[174] J. Slein, E. J. Whitehead, Jr., J. Davis, G. Clemm, C. Fay, and J. Crawford, “WebDAV Bindings,” Xerox, U.C. Irvine, CourseNet, Rational, FileNet, IBM. Internet-Draft, work-in-progress, December 17, 1999.

[175] J. A. Slein, F. Vitali, E. J. Whitehead, Jr., and D. Durand, “Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web,” Xerox, Univ. of Bologna, U.C. Irvine, Boston Univ. Internet Information Request for Comments (RFC) 2291, February, 1998.

[176] J. B. Smith and F. D. Smith, “ABC: A Hypermedia System for Artifact-Based Collaboration,” *Proc. Third ACM Conference on Hypertext (Hypertext'91)*, San Antonio, Texas, Dec. 15-18, 1991, pp. 179-192.

[177] J. B. Smith, S. F. Weiss, and G. J. Ferguson, “A Hypertext Writing Environment and its Cognitive Basis,” *Proc. ACM Hypertext'87 Workshop*, Chapel Hill, NC, Nov. 13-15, 1987, pp. 195-214.

[178] L. F. G. Soares, G. L. d. S. Filho, R. F. Rodrigues, and D. Muchaluat, “Versioning Support in the HyperProp System,” *Multimedia Tools and Applications*, vol. 8, no. 3 (1999), pp. 325-339.

[179] L. F. G. Soares, N. L. R. Rodriguez, and M. A. Casanova, “Nested Composite Nodes and Version Control in an Open Hypermedia System,” *International Journal on Information Systems (Special issue on Multimedia Information Systems)*, vol. 20, no. 6 (1995), pp. 501-520.

[180] I. Sommerville, T. Rodden, P. Rayson, A. Kirby, and A. Dix, “Supporting information evolution on the WWW,” *World Wide Web*, vol. 1, no. 1 (1998), pp. 45-54.

[181] N. Streitz, “SEPIA: A Cooperative Hypermedia Authoring Environment,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 11-22.

[182] W. S. Strong, *The Copyright Book*, Fifth ed. Cambridge, MA: MIT Press, 1999.

[183] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, “Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems,” *ACM Transactions on Human-Computer Interaction*, vol. 5, no. 1 (1998), pp. 63-108.

[184] R. N. Taylor, W. Tracz, and L. Coglianese, “Software Development Using Domain-Specific Software Architectures: A Curriculum Module in the SEI Style,” Technical Report ADAGE-UCI-94-01C, 1994.

[185] W. F. Tichy, “RCS - A System for Version Control,” *Software-Practice and Experience*, vol. 15, no. 7 (1985), pp. 637-654.

[186] W. F. Tichy, “Tools for Software Configuration Management,” *Proc. International Workshop on Software Version and Configuration Control*, Grassau, Germany, January, 1988, pp. 1-20.

[187] W. Tracz, *Confessions of a used program salesman: Institutionalizing software reuse*. Reading, MA: Addison-Wesley, 1995.

[188] W. Tracz, “DSSA (Domain-Specific Software Architecture),” *Software Engineering Notes*, vol. 20, no. 3 (1995), pp. 49-62.

[189] R. Trigg, L. Suchman, and F. Halasz, “Supporting Collaboration in NoteCards,” *Proc. Computer Supported Cooperative Work (CSCW'86)*, Austin, Texas, Dec. 3-5, 1986, pp. 147-153.

[190] R. H. Trigg, “Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment,” *ACM Trans. on Office Information Systems*, vol. 6, no. 4 (1988), pp. 398-414.

[191] R. H. Trigg and M. Weiser, “TEXTNET: A Network-Based Approach to Text Handling,” *ACM Trans. on Office Information Systems*, vol. 4, no. 1 (1986), pp. 1-23.

[192] A. van der Hoek, “A Generic Peer-to-Peer Repository for Distributed Configuration Management,” *Proc. 18th Int'l Conference on Software Engineering (ICSE-18)*, Berlin, Germany, March, 1996, pp. 308-317.

[193] F. Vitali, “Versioning Hypermedia,” *ACM Computing Surveys*, vol. 31, no. 4 (1999).

[194] F. Vitali and D. G. Durand, “Using Versioning to Support Collaboration on the WWW,” *Proc. Fourth International World Wide Web Conference*, Boston, MA, November, 1995, pp. 37-50.

[195] L. Wakeman and J. Jowett, *PCTE: The Standard for Open Repositories*. New York: Prentice Hall, 1993.

[196] D. W. Weber, “CM Strategies for RAD,” *Proc. Ninth Int'l Symposium on System Configuration Management (SCM-9)*, Toulouse, France, Sept. 5-7, 1999, pp. 204-216.

[197] B. Westfechtel and R. Conradi, “Software Configuration Management and Engineering Data Management: Differences and Similarities,” *Proc. System Configuration Management (SCM-8)*, Brussels, Belgium, July 20-21, 1998, pp. 95-106.

[198] E. J. Whitehead, Jr., “An Architectural Model for Application Integration in Open Hypermedia Environments,” *Proc. The Eighth ACM Conference on Hypertext, Hypertext'97*, Southampton, UK, April 6-11, 1997, pp. 1-12.

[199] E. J. Whitehead, Jr., “Goals for a Configuration Management Network Protocol,” *Proc. 9th Int'l Symposium on System Configuration Management (SCM-9)*, Toulouse, France, Sept. 5-7, 1999, pp. 186-203.

[200] E. J. Whitehead, Jr., K. M. Anderson, and R. N. Taylor, “A Proposal for Versioning Support for the Chimera System,” *Proc. Workshop on Versioning in Hypertext Systems*, Edinburgh, Scotland (held with ECHT'94), Sept. 18-23, 1994, pp. 45-54.

[201] E. J. Whitehead, Jr. and Y. Y. Goland, “WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web,” *Proc. Sixth European Conference on Computer Supported Cooperative Work*, Copenhagen, Denmark, Sept. 12-16, 1999, pp. 291-310.

[202] U. K. Wiil and J. J. Leggett, “Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems,” *Proc. Fourth ACM Conference on Hypertext (ECHT'92)*, Milano, Italy, Nov. 30-Dec. 4, 1992, pp. 251-261.

[203] U. K. Wiil and J. J. Leggett, “Concurrency Control in Collaborative Hypertext Systems,” *Proc. Hypertext'93*, Seattle, WA, Nov. 14-18, 1993, pp. 14-24.

[204] U. K. Wiil and J. J. Leggett, “The HyperDisco Approach to Open Hypermedia Systems,” *Proc. Seventh ACM Conference on Hypertext (Hypertext '96)*, Washington, DC, March 16-20, 1996, pp. 140-148.

[205] N. Yankelovich, B. J. Haan, N. K. Meyrowitz, and S. M. Drucker, “Intermedia: The Concept and the Construction of a Seamless Information Environment,” *IEEE Computer*, vol. 21, no. 1 (1988), pp. 81-96.