UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND APPLIED SCIENCE

School of Electronics and Computer Science

The Object and Connection Space Approach To Opening Up Hypermedia Structure

by

Jon-Paul Griffiths, B.Sc.

Thesis for the degree of Doctor of Philosophy

May 2005

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

THE OBJECT AND CONNECTION SPACE APPROACH TO OPENING UP HYPERMEDIA STRUCTURE

by Jon-Paul Griffiths

Open hypermedia emphasises the separation of hypermedia structure and data, typically achieved through the removal of hypermedia mark-up within data. Such removal enables re-application of the same hypermedia structure to different sets of data.

However, the internal organisation of hypermedia structure does not follow these same separation rules as most hypermedia systems continue to embed mark-up within hypermedia structure. This violates the principles of open hypermedia since the objects and connections of a hypermedia structure cannot be re-used within other hypermedia structures. Hence only entire hypermedia structure re-use is possible. This is not only an inefficient use of resources, but it highlights the anomaly that the objects and connections of open hypermedia structure are not actually open to re-use.

This thesis describes the Object and Connection Space (OCS) data model. It opens up hypermedia structure by separating the functional and connectional aspects of hypermedia structure. Once assigned to Object and Connection Spaces the productive re-use of hypermedia structure can begin, either as individual hypermedia objects or as collections of selected hyperstructure segments. The thesis also explores how the OCS data model, via structure re-use, enables improved structure versioning, in particular the prevention of revision proliferation. And it considers how the OCS data model benefits the overall maintenance of hypermedia structure by offering a more logical approach to the repair of broken internal routes within hypermedia structure.

Contents

Contents i
List of Figures ix
List of Tables xiii
Declaration of Authorship xiv
Acknowledgements xv
Chapter 1. Introduction 1
1.1. Internal Organisation of Hypermedia Structure 1
1.2. The Object and Connection Space Approach 4
1.3. Contributions to Computer Science 4
1.4. Thesis Structure 5
Chapter 2. Hypermedia Background 7
2.1. Introduction 7
2.2. The Promise of Hypermedia 7
2.3. Modern Hypermedia Systems 7
2.4. Hypermedia Structure 8
2.5. Re-use in Hypermedia 9
2.6. Elementary Composition of a Hypermedia System 10
2.7. The Early Years 11
2.7.1. Bush 12
2.7.2. Engelbart 13
2.7.3. Nelson 13
2.7.3.1. Preventing Duplication of Re-used Document Content 15
2.7.3.2. Highlighting Re-used Document Revisions 15
2.7.3.3. Clickable Re-used Content Segments 16
2.8. Monolithic Systems 18
2.9. Client-Server Hypermedia Systems 19
2.10. The World Wide Web 20
2.11. The Dexter Hypertext Reference Model 22
2.12. Open Hypermedia Systems 23
2.12.1. Similarities between Different Open Hypermedia Systems 24
2.12.2. Microcosm 24
2.12.3. Re-use in Open Hypermedia Systems 26
2.12.4. A Lack of Consensus 26
2.13. The Open Hypermedia Protocol Project 28
2.14. Multiple Open Systems 28

2.15. Structural Computing 29
2.15.1. Construct 31
2.16. The Fundamental Open Hypermedia Model 32
2.17. Summary 33
Chapter 3. OHP and FOHM 34
3.1. Introduction 34
3.2. The Open Hypermedia Protocol 34
3.2.1. Core Architecture 35
3.2.2. Core Data Model 36
3.2.2.1. Collaboration Classes 38
3.2.2.2. Computation Classes 38
3.2.3. OHP-Nav Protocol 39
3.2.4. Condensed OHP-Nav Data Model 41
3.2.5. Linked List Representation 43
3.2.6. OHP-Nav Traversal 45
3.2.6.1. Carrying Out Link Traversal 45
3.2.6.2. Retrieving Individual OHP-Nav Objects 46
3.2.7. Implementations 47
3.2.7.1. Example OHP-Nav message 48
3.2.8. OHP Demonstrations 50
3.2.8.1. Hypertext '98 Demonstration 50
3.2.8.2. Hypertext '99 Demonstration 51
3.3. The Fundamental Open Hypermedia Model 52
3.3.1. FOHM Background 52
3.3.2. Hypermedia Domains 53
3.3.2.1. Navigational Hypermedia Domain 53
3.3.2.2. Spatial Hypermedia Domain 53
3.3.2.3. Taxonomic Hypermedia Domain 54
3.3.3. FOHM and Structural Computing 54
3.3.4. The FOHM Data Model 55
3.4. Summary 58
Chapter 4. Versioning Background 59
4.1. Introduction 59
4.2. What is Versioning? 59
4.3. Versioning Policy 59
4.4. Why Version Hypermedia Resources? 60
4.5. Hypermedia Versioning Systems 60
4.5.1. Design Spaces for Versioning 61

4.5.2. Xanadu ... 62
4.5.3. HyperPro ... 62
4.5.4. CoVer ... 63
4.5.5. DeltaV ... 64
4.5.6. HyperProp ... 64
4.5.7. Hypermedia Versioning Control Framework ... 65
4.5.8. Chimera Versioning ... 66
4.6. Revision Proliferation ... 66
4.6.1. HyperPro ... 67
4.6.2. CoVer ... 67
4.6.3. HyperProp ... 70
4.7. Summary ... 71
Chapter 5. Referential Integrity Background ... 73
5.1. Introduction ... 73
5.2. Importance of Referential Integrity ... 73
5.3. Forms of Broken Hypermedia Structure ... 74
5.3.1. Dangling Hypermedia Structures ... 74
5.3.2. Specious Hypermedia Structures ... 74
5.3.3. Misaligned Internal References ... 74
5.3.4. Broken Internal Routes ... 75
5.4. Link Repair Strategies ... 76
5.4.1 Passive Approach ... 76
5.4.2. User Onus ... 76
5.4.3. Tightly Coupled Link Repair ... 77
5.4.4. System Tools for Pro-Active Users ... 77
5.4.4.1. Forward References ... 77
5.4.4.2. Link Integrity Checkers ... 78
5.4.5. Just-In-Time Link Repairs ... 79
5.4.6. Preventing Broken Links Altogether ... 79
5.4.6.1. Publishing Model ... 79
5.4.6.2. Hypertext Link Queries ... 79
5.4.6.3. Declarative Links ... 80
5.4.6.4. Dynamic Links ... 80
5.4.7. Prevention via Versioning ... 81
5.4.7.1. Versioning Hypermedia Documents ... 81
5.4.7.2. Versioning Hypertext Links ... 82
5.4.7.3. Versioning Hypermedia Documents and Hypertext Links ... 82
5.5. Summary ... 83

Chapter 6. Problem Domain Issues 84
6.1. Introduction 84
6.2. Issue One: Repetitive Hypermedia Objects 84
6.2.1. Repetitive Hypermedia Objects in OHP-Nav 84
6.2.2. Repetitive Hypermedia Objects in FOHM 87
6.2.3. Dynamic Linking 89
6.2.4. Repetitive Hypermedia Object Problem 90
6.3. Issue Two: Repetitive Hypermedia Structure 90
6.3.1. Repetitive Hypermedia Structures in OHP-Nav 90
6.3.2. Repetitive Hypermedia Structures in FOHM 93
6.3.3. Repetitive Hypermedia Structure Problem 96
6.4. Issue Three: Revision Proliferation 97
6.4.1. The OHP-Version Protocol 97
6.4.2. Contextualised Connections 101
6.4.2.1. Adjustments to the FOHM Data Model 101
6.4.2.2. Contextualising FOHM Connections 102
6.4.2.3. Limiting Revision Proliferation 103
6.4.2.4. Embedding Still Causes Revision Proliferation 104
6.5. Issue Four: Hypermedia Structure Maintenance 105
6.5.1. Multiple Internal Routes Within Structure 106
6.5.2. Broken Internal Routes 106
6.5.3. A Confusing Repair Process 107
6.6. Summary 109
Chapter 7. Opening Hypermedia Structure 110
7.1. Introduction 110
7.2. Unopen Hypermedia Structure 111
7.3. OCS Data Model Objectives 112
7.4. C-Level Work 113
7.5. Drawing on OHP-Nav 113
7.6. "Rock, Paper, Scissors" 113
7.7. The Object and Connection Space Data Model 116
7.7.1. The Function Object Space 117
7.7.2. The Connection Space 118
7.7.2.1. Binary Connection Objects 118
7.7.2.2. N-ary Connection Objects 119
7.7.2.3. Which to Choose: Binary or N-ary Connections? 119
7.7.3. Comparison with OHS Hypertext Links 121
7.8. Connection Objects Explained 124

7.9. Lightweight vs. Heavyweight Objects 126
7.10. Connecting Connection Objects 127
7.10.1. Conjoinment Operation 127
7.10.2. Attachment Operation 127
7.10.3. Linking To Hypermedia Structure 128
7.11. Issues 128
7.11.1. Anchoring 128
7.11.2. Object Uniqueness 130
7.11.2.1. Re-used Function Objects within a Single Connection Object 130
7.11.2.2. Re-used Function Objects in External Connection Objects 132
7.12. XML Specification 134
7.13. Class and Instance Relationship 136
7.14. Object and Connection Space Example 137
7.15. Impact on the OHS Architecture 140
7.15.1. Client Application Layer 140
7.15.2. Middleware Layer 141
7.15.3. Storage Back End Layer 141
7.16. Added Complexity 141
7.16.1. Separating and Building Structure 142
7.16.2. Re-using Structure 143
7.17. Summary 143
Chapter 8. Implications for OHP-Nav and FOHM 145
8.1. Introduction 145
8.2. Embedding within Hypermedia Objects 146
8.3. Application of the OCS Data Model 147
8.4. Resolving Problem Domain Issues 1 and 2 148
8.4.1. OHP-Nav Single Hypermedia Object Re-use 148
8.4.2. FOHM Repetitive Hypermedia Structure Re-use 151
8.5. Imitating OHP-Nav Structural Organisation 154
8.6. Applying the OCS Data Model to Connection Objects 156
8.6.1. Why Change the Structure Representation of Connection Objects? 157
8.6.2. Transforming Connection Objects to FOHM Structures 159
8.6.3. Applying the OCS Data Model to FOHM Structures 161
8.6.4. Scope of the OCS Data Model 166
8.7. Summary 168
Chapter 9. Applications for Versioning 169
9.1. Introduction 169
9.2. Recap on Revision Proliferation 169

9.3. OCS Solution to Revision Proliferation ... 170
9.4. Different Views of Hypermedia Structure ... 173
9.5. OCS Data Model vs. Contextualised Connections ... 175
9.6. Summary ... 177
Chapter 10. A Versioning Framework ... 179
10.1. Introduction ... 179
10.2. Versioning Framework ... 180
10.3. Versioning Framework Organisation ... 180
10.4. Versioning Example ... 181
10.5. OCS Framework Advantages ... 185
10.5.1. Connection Integrity ... 185
10.5.2. Framework Alterations ... 185
10.5.3. Easier Connections ... 187
10.6. OCS Solutions to Existing Revision Proliferation Problems ... 187
10.6.1. CoVer ... 187
10.6.2. The Nested Composite Model ... 190
10.7. Summary ... 192
Chapter 11. Applications for Link Maintenance ... 194
11.1. Introduction ... 194
11.2. Benefiting Link Maintenance ... 194
11.3. A More Logical Approach ... 195
11.4. Versioned Hypertext Links ... 197
11.5. Finding Connections Between Objects ... 198
11.6. Summary ... 200
Chapter 12. Conclusions ... 202
12.1. Introduction ... 202
12.2. Restatement of the Problem ... 203
12.3. The OCS Data Model ... 203
12.4. Scope of the OCS Data Model ... 204
12.5. Problem Domain Issues Answered ... 205
12.5.1 Issue 1: Repetitive Hypermedia Objects ... 205
12.5.2. Issue 2: Repetitive Hypermedia Structure ... 206
12.5.3. Issue 3: Revision Proliferation ... 207
12.5.4. Issue 4: Improved Hypermedia Structure Maintenance ... 208
12.6. Computer Science Contributions ... 208
12.6.1. CSC1: Extending the Concept of Open Hypermedia ... 208
12.6.2. CSC2: Promoting the general re-use of hypermedia structure ... 209
12.6.3. CSC3: Logical approach to hyperstructure representation ... 210

12.6.4. CSC4: Improved versioning of hypermedia structure 210
12.6.5. CSC5: Improved hypermedia structure maintenance 211
12.7. Relationship to Existing Research 211
12.7.1. Xanadu 212
12.7.2. The World Wide Web 213
12.7.3. Open Hypermedia 213
12.7.4. Structural Computing 215
12.7.5. Object Prototyping 216
12.7.6. Standard CB-OHS Storage Interface 217
12.7.7. Callimachus 218
12.7.8. Themis 220
12.8. Future Work 221
12.8.1. Implementation 221
12.8.2. Data Models 221
12.8.3. Hypermedia Domains 222
12.8.4. The OCS Framework 222
12.8.5. Standardisation of Connection Objects 222
12.8.6. Link Maintenance Repair Applications 223
12.9. Overall Conclusion 223
Appendix A. OCS Representation in XML 224
A.1. Introduction 224
A.2. Conventional OHP-Nav Primary Objects 224
A.3. OCS Function Objects 226
A.4. OCS Connection Objects 227
A.5. Re-use Example 229
A.5.1. Function Object Space 230
A.5.2. Connection Object Space CS500 232
A.5.3. Connection Object Space CS501 235
A.6. Converting to OCS Data Model Format Example 240
A.6.1. XML for Conventional Hypertext Link 241
A.6.2. XML for OCS Function Objects 244
A.6.3. XML for OCS Connection Objects 246
Appendix B. Examples of OCS Re-use 249
B.1. Introduction 249
B.2. FOHM Single Hypermedia Object Re-use 249
B.2.1. OCS Solution with FOHM Object Embedding 249
B.2.2. OCS Solution without Object Embedding 251
B.3. OHP-Nav Repetitive Hypermedia Structure Re-use 254

B.3.1. 'Cats Eat' Something Relationship 254
B.3.2. Node-less 'Eat' Relationship 256
References 261

List of Figures

Figure 1.1: Hypertext link composed of hypermedia objects and connections......... 2
Figure 1.2: Hypermedia objects containing embedded connection data. 3
Figure 1.3: Example application of the OCS data model... 4
Figure 2.1: Hypermedia structure stored separately in a link database.................. 11
Figure 2.2: Evolution of hypermedia systems.. 11
Figure 2.3: What the memex may have looked like 13
Figure 2.4: Xanadu permascroll scenario.. 14
Figure 2.5: Transclusion links between different document revisions 16
Figure 2.6: Transclusion link showing re-use between two documents 17
Figure 2.7: Permascroll scenario depicting the transclusion link of Figure 2.6 18
Figure 2.8: Dexter Hypertext Reference Model.. 23
Figure 2.9: Microcosm architecture... 25
Figure 2.10: Structural Computing hypermedia structure service provision............ 30
Figure 2.11: Construct CB-OHS architecture... 31
Figure 3.1: Common Reference Architecture data model 35
Figure 3.2: OHSWG Core Data Model 37
Figure 3.3: Navigation Classes of the OHSWG Core Data Model 40
Figure 3.4: Condensed OHP-Nav data model.. 42
Figure 3.5: Common representation of an OHP-Nav hypertext link........................ 43
Figure 3.6: Amended representation of OHP-Nav hypermedia objects 44
Figure 3.7: Generic OHP-Nav OHS environment 45
Figure 3.8: Conceptual architecture of the Solent CB-OHS....................................... 47
Figure 3.9: OHP-Nav message to create an Anchor object.. 49
Figure 3.10: Example Taxonomy... 54
Figure 3.11: Auld Linky within a Structural Computing environment...................... 55
Figure 3.12: Example FOHM structure 57
Figure 4.1: Versioning HyperPro contexts.. 63
Figure 4.2: User Context Nodes of the HyperProp Nested Composite Model 65
Figure 4.3: HyperPro revision proliferation scenario... 67
Figure 4.4: CoVer revision proliferation example. 68
Figure 4.5: CoVer revision proliferation solution... 69
Figure 4.6: Revision proliferation in NCM 70
Figure 4.7: Propagation Guided By Perspective. 71
Figure 5.1: Hypertext link containing two internal routes... 75
Figure 6.1: OHP-Nav hyperstructures containing identical hypermedia objects ... 85

Figure 6.2: Node re-use in OHP-Nav 87
Figure 6.3: Two FOHM hyperstructures containing identical objects 88
Figure 6.4: Example of a repetitive segment in two hypermedia structures 91
Figure 6.5: The wrong structural organisation: 'Cats Eat Meat and Fish' 92
Figure 6.6: FOHM hyperstructure containing repetitive structure 94
Figure 6.7: Incorrect re-use of hyperstructure segment 95
Figure 6.8: Possible FOHM structure segment re-use 96
Figure 6.9: Initial and expected OHP-Nav hypermedia structures 98
Figure 6.10: OHP-Nav hyperstructure after OHP-Version application 98
Figure 6.11: FOHM structure before and after revision to enable versioning 102
Figure 6.12: Example of a contextualised connection 103
Figure 6.13: Adding a new revision to an existing FOHM structure during the Contextual Connection process 104
Figure 6.14: Contextualised Connections causing revision proliferation 105
Figure 6.15: Hypertext link containing two internal routes 106
Figure 6.16: Repaired hypertext link 107
Figure 7.1: 'Rock, Paper, Scissors' outcomes 114
Figure 7.2: 'Rock, Paper, Scissors' hyperstructure outcomes. 115
Figure 7.3: Failed RPS hypermedia object re-use prior to application of the OCS data model 116
Figure 7.4: Hypermedia structure: 'Rock defeats Scissors' 117
Figure 7.5: Example of a Function Object Space 117
Figure 7.6: Connection Space of binary Connection Objects 119
Figure 7.7: Connection Space of binary and n-ary Connection Objects 119
Figure 7.8: Example of connections being re-used 120
Figure 7.9: Comparison of OHS node and links with OCS Function and Connection Objects 122
Figure 7.10: Example of an n-ary OHS hypertext link 123
Figure 7.11: Two examples of n-ary Connection Objects 124
Figure 7.12: Example of a hypertext network 124
Figure 7.13: Conjoin operation 127
Figure 7.14: Attachment operation 128
Figure 7.15: How should a Function Object be attached to another Function Object inside a Connection Object? 129
Figure 7.16: How a Function Object attaches to another Function Object inside a Connection Object 130
Figure 7.17: Desired re-use of a Function Object 131
Figure 7.18: Undesirable re-use of a Function Object 131

Figure 7.19: Instance Objects 132
Figure 7.20: Attaching externally re-used Function Objects together 133
Figure 7.21: Assigning Instance identifiers to Connection Objects 134
Figure 7.22: XML example of a conventional OHP-Nav Endpoint object and OCS Endpoint Function Object 135
Figure 7.23: XML code to create a Connection Object 136
Figure 7.24: 'Paper defeats Rock' relationship 138
Figure 7.25: Object and Connection Spaces to represent RPS relationships 138
Figure 7.26: Incorrect RPS hyperstructure without OCS approach 139
Figure 8.1: OCS solution for the OHP-Nav hyperlinks of Figure 6.1 149
Figure 8.2: OCS solution for the FOHM structure of Figure 6.6 152
Figure 8.3: FOHM hyperstructure containing a re-used structure segment 154
Figure 8.4: Imitating OHP-Nav hypermedia structure representation 155
Figure 8.5: OHP-Nav 'Humans Eat Fruit' hyperstructure 156
Figure 8.6: OHP-Nav structure of Figure 8.5 assigned to the OCS data model 157
Figure 8.7: Ineffectual application of OCS data model to a Connection Object 158
Figure 8.8: Transformation of Connection Objects to FOHM structures 160
Figure 8.9: Application of the OCS data model to the Connection Objects of Connection Space CS110 of Figure 8.6 162
Figure 9.1: Original and new hyperstructures after new object revision created . 170
Figure 9.2: Object and Connection Spaces for Figure 9.1 171
Figure 9.3: More Connection Spaces for Figure 9.1 172
Figure 9.4: Independent structures as OHP-Nav structure revisions 175
Figure 9.5: FOHM Contextualised Connections as Connection Objects 177
Figure 10.1: The OCS Versioning Framework 181
Figure 10.2: Original and new hyperstructures after new object revision created . 182
Figure 10.3: Connection Objects for hyperstructure revisions 182
Figure 10.4: Function Object Revision Information 182
Figure 10.5: Function Object revision history 183
Figure 10.6: Connection Object revision history 184
Figure 10.7: Object Space for Function and Connection Object Revision History .. 184
Figure 10.8: Re-organised Versioning Framework 186
Figure 10.9: Object and Connection Spaces for CoVer versioning example 188
Figure 10.10: Connection Spaces recording CoVer revision information 189
Figure 10.11: Connection Spaces recording CoVer evolution history 190
Figure 10.12: Desired solution for NCM revision proliferation problem 191
Figure 10.13: Object and Connection Spaces for NCM solution 192
Figure 11.1: Before repair – OCS depiction of hypertext link of Figure 6.15 195

Figure 11.2: After repair – OCS depiction of hypertext link of Figure 6.16 196
Figure 11.3: Locating connections within OHP-Nav vs. the OCS data model 199
Figure A.1: OCS Function Object Space for all RPS 'defeats' scenarios 229
Figure A.2: Connection Space CS500 230
Figure A.3: Connection Space facilitating all RPS 'Defeats' scenarios 230
Figure A.4: Conventional hypertext link representation 240
Figure A.5: OCS Function Objects imitating OHP-Nav objects 240
Figure A.6: OCS Connection Objects imitating OHP-Nav connections 241
Figure B.1: OCS solution enabling single FOHM object re-use for the FOHM structure of Figure 6.3 permitting wholesale object embedding 250
Figure B.2: OCS solution enabling single FOHM object re-use for the FOHM structure of Figure 6.3 without wholesale object embedding 252
Figure B.3: OCS solution for the 'Cats Eat' relationship of Figure 6.4 254
Figure B.4: Graphical depiction of the OCS solution of Figure B.3 256
Figure B.5: OCS solution for OHP-Nav Node-less relationship of Section 6.3.1 ... 257
Figure B.6: OHP-Nav embedded object reference hypermedia structure of 'Birds Eat Worms' relationship 258
Figure B.7: Graphical depiction of the OCS solution of Figure B.5. 260

List of Tables

Table 1.1: Summary of functions performed by objects of Figure 1.1 3
Table 6.1: Hypermedia objects of Figure 6.1 performing the same function 85
Table 6.2: Revision proliferation caused by creating new revision 'A2 v2' 100
Table 7.1: Discovering new node object resource relationships 140
Table 8.1: Summary of re-use within the OCS solution of Figure 8.1 150
Table 8.2: Summary of re-use within the OCS solution of Figure 8.2 153
Table 8.3: Imitating OHP-Nav relationships 155
Table 8.4: Summary of re-use within the OCS solution of Figure 8.9 165
Table B.1: Summary of re-use within the OCS solution of Figure B.1................... 251
Table B.2: Summary of re-use within the OCS solution of Figure B.2................... 253
Table B.3: Summary of re-use within the OCS solution of Figure B.3................... 255
Table B.4: Summary of re-use within the OCS solution of Figure B.5................... 259

Acknowledgements

I would like to dedicate this thesis in loving memory to my Father, Vic, and my Grandparents, Tom and Cath.

Special thanks must go to my supervisor, Hugh Davis. I will be forever grateful for his endless support and advice which was of enormous help. In particular ploughing through the many drafts of my thesis. His insight into the world of hypermedia proved to be most invaluable.

I must also thank Dr David Millard. He was a first-rate partner when working on hypermedia implementations. Discussions about the OCS and OHP were always informative and very beneficial. And most importantly I must thank him for proof reading this thesis.

I would also like to thank Dr Sigfried Reich. He proved to be a fountain of knowledge, and was always prepared to drop everything to lend a helping hand. Of course I need to thank all the members of the Southampton IAM research group for their help along as well as all the members of the OHSWG (Open Hypermedia Systems Working Group).

I also need to express my thanks to a special group of friends, Malcolm, Graham, Evan, Nigel, Jed and Jenny, who have just about kept me sane during this journey.

Finally I would like to pay special tribute to my parents, Vic and Margaret. Their love, support and encouragement over the years has been an enormous source of strength.

And now, at long last, I can finally say I've done it!

Chapter 1. Introduction

The emphasis of open hypermedia is on the separation of hypermedia structure and data. This is achieved through removing the hypermedia mark-up within data. Such mark-up comprises hypertext links and/or embedded anchors. The result is hypertext links are stored as independent entities in link databases. Examples of systems that practice such open hypermedia include Microcosm [Davis et al. 1992], DHM [Grønbæk et al. 1994] and HyperDisco [Wiil and Leggett 1996].

Such an open scheme offers advantages. Because hypermedia documents are not embedded with hypermedia structure means hypermedia documents can be re-used to be associated with altogether different sets of documents. Hence it is possible to produce different views of the same hypertext network. Moreover, storing hypermedia structure (in the form of hypertext links) as independent entities enables that same hypermedia structure to be applied to different sets of data. This is not only an efficient use of resources, but it helps with the discovery of new relationships between documents (e.g. Microcosm generic links [Davis et al. 1992]).

Yet despite the advantages listed (and more besides), the internal organisation of hypermedia structure does not follow these same separation rules.

1.1. Internal Organisation of Hypermedia Structure

The internal organisation of hypermedia structure is composed of hypermedia objects and hypermedia structural connections. Figure 1.1 shows an example. Hypermedia objects provide the functional information about the structure, such as the identity of the node(s) referenced by the structure or the direction that the structure should be followed. The structural connections express which hypermedia objects are connected to one another.

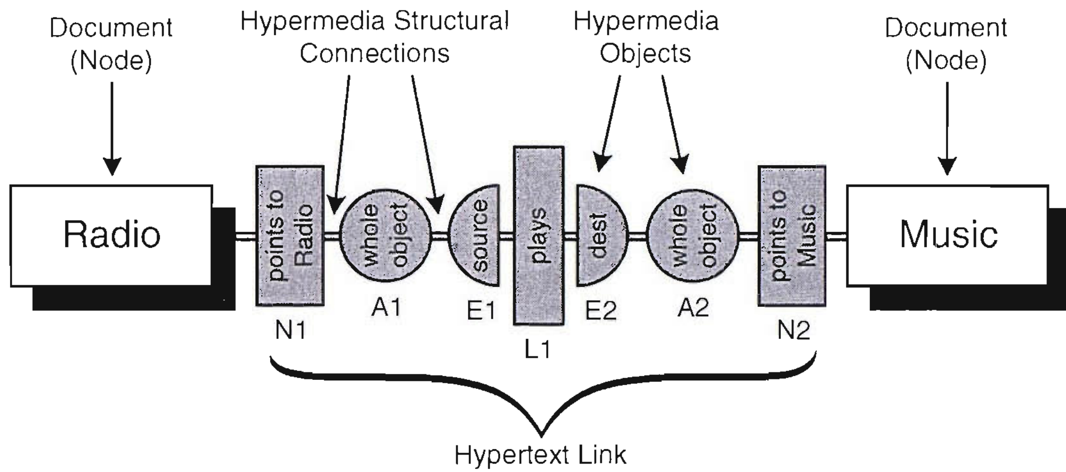


Figure 1.1: Hypertext link composed of hypermedia objects and connections linking two documents together.

The role of the hypertext link of Figure 1.1 is to establish the 'plays' relationship between the 'Radio' and 'Music' documents in order to make the association that 'Radio plays Music'. Table 1.1 summarises the functions being performed by the seven hypermedia objects of the hypertext link.

Unlike the case with open hypermedia which removes all embedded mark-up from node data, hypermedia structure continues to embed structural mark-up within individual hypermedia objects. This usually takes the form of embedding connection data within hypermedia objects. Figure 1.2 shows a simple example of such a scenario. The blocks at each end of a connection represent the embedded mark-up within an object, and the arrowhead (at the other end of the connection) points to the connected object. When taken together, this mark-up (and only this mark-up) demarcate that there is a connection between both objects. There is no separation between hypermedia object and connection.

ID	Label Name	Hypermedia Object Description
N1	points to Radio	Declares to which node (in this case the 'Radio' document) the hypertext link is referencing.
A1	whole object	Indicates that the hypertext link is referencing the whole ('Radio' document) node.
E1	source	Indicates that the 'Radio' document is the source of the hypertext link.
L1	plays	Describes the nature of the relationship between the two connected nodes.
E2	dest	Indicates that the 'Music' document is the destination of the hypertext link.
A2	whole object	Indicates that the hypertext link is referencing the whole ('Music' document) node.
N2	points to Music	Declares to which node (in this case the 'Music' document) the hypertext link is referencing.

Table 1.1: Summary of the functions being performed by the hypermedia objects of the hypertext link of Figure 1.1.

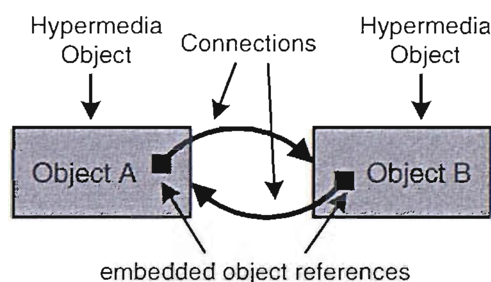


Figure 1.2: Hypermedia objects containing embedded connection data.

Such an embedded approach to the internal organisation of hypermedia structure violates the principles of open hypermedia as it prevents the objects and connections of a hypermedia structure from being re-used within other hypermedia structures. Only entire hypermedia structure re-use is possible. This is not only an inefficient use of resources, but it highlights the anomaly that the objects and connections of open hypermedia structure are *not* indeed open.

To this end I devised the Object and Connection Space (abbreviated to OCS) approach. It provides a data model and methodology that defines how hypermedia

objects may be connected together in order to build hypermedia structure. It is this data model that is the subject of the thesis.

1.2. The Object and Connection Space Approach

The role of the OCS data model is to conceptually re-structure the internal organisation of hypermedia structure. This is achieved by separating hypermedia structure into hypermedia objects and hypermedia connections. Figure 1.3 shows a graphical example of the OCS approach being applied to the hypermedia structure of Figure 1.2.

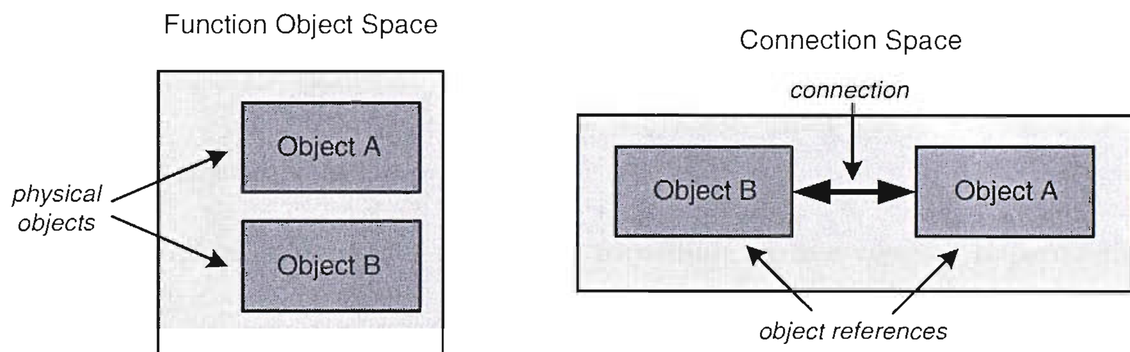


Figure 1.3: Example application of the OCS data model.

Objects A and B are stored in the Function Object Space minus their connection information. The connection data is stored in the Connection Space. It identifies which objects are connected together using references to objects rather than storing the physical objects themselves.

The reasons for and the advantages gained by separating hypermedia structure into separate hypermedia objects and hypermedia connections are the subject of the thesis.

1.3. Contributions to Computer Science

My contribution to Computer Science is the Object and Connection Space approach to internally organising the structure of hypermedia structure.

The OCS approach has the following advantages:

- It extends the concept of open hypermedia into the realm of hypermedia structure.
- It promotes the re-use of hypermedia structure.
- Its data model offers a more logical approach to hypermedia structure representation.
- It enables improved versioning of hypermedia structure.
- It enables improved hypermedia structure maintenance.

1.4. Thesis Structure

The thesis is structured as follows.

Chapter 2 defines hypermedia as well as providing an overview of hypermedia history.

Chapter 3 describes OHP-Nav and FOHM. It was these two hypermedia structure data models that led to the development of the Object and Connection Space data model. It was on their implementation that alerted the need to re-define the organisation of the underlying data model for hypermedia structure.

Chapter 4 provides an overview of versioning within the hypermedia field as the OCS data model has a direct impact on hypermedia versioning.

Likewise *Chapter 5* examines current approaches to hypertext link maintenance since the OCS approach also has a bearing on improved hypertext link integrity.

Chapter 6 is a critical chapter as it sets out the specific problems that the OCS data model is addressing.

Chapter 7 begins the discussion of the OCS data model by describing the basic concepts behind the OCS methodology.

Chapter 8 explores the implications of applying the OCS data model to the existing OHP-Nav and FOHM data models.

Chapter 9 explains how the OCS data model can benefit the versioning of hypermedia structure.

Chapter 10 demonstrates how the principles of the OCS representation can be used as a framework for constructing different hypermedia structure relationships.

Chapter 11 explains how the OCS data model can improve hypertext link maintenance.

And finally *Chapter 12* offers conclusions and suggests further work.

Chapter 2. Hypermedia Background

2.1. Introduction

This chapter explores the concept of hypermedia along with a potted history of the evolution of hypermedia systems. Concurrent to this is a commentary on how re-use has played a part in shaping hypermedia and hypermedia system development.

2.2. The Promise of Hypermedia

Hypermedia[1] offers radically different ways of structuring information. It is not only an efficient tool for organising large collections of unstructured information, but it can also be used for the more grandiose task as an idea processing tool. This is because people think in terms of ideas, facts and evidence, not in terms of screenfuls of text. Hypermedia reflects these human processes. Ideas can be expressed within independent objects referred to as nodes. These can be linked, moved and changed as the thought processes of the writer evolve. New ideas can be developed within new nodes and linked to existing ideas. Moreover individual ideas are not fixed within a rigid structure but they can be re-used by being referenced elsewhere. Nodes can be used to construct flexible networks that model a problem or solution where links form the glue that binds the nodes together.

2.3. Modern Hypermedia Systems

Modern hypermedia systems reside exclusively within the domain of digital computers. They act as the interface enabling users to interact directly with chunks of data (nodes) and to establish new relationships (links) between them. They can be used to create, annotate, link together and share information from a variety of media, including text, graphics, audio and programs.

[1] The terms hypermedia and hypertext are used interchangeably. Strictly speaking hypertext is the same as hypermedia but restricted to text information only, i.e. hypermedia without the multimedia component.

Hypertext links can point to whole nodes (e.g. documents) or specific regions within a node. Nodes can be any size; and link-endpoints (that point to nodes) can mark the node as being source or destination. This describes whether the node is the beginning or ending point for the hypertext link. Nodes often contain link markers, called anchors, which when clicked activate other nodes. Hypertext links can be uni-directional (enabling one-way traversal) or bi-directional (facilitating backward traversal as well). They can be referential (for cross-referencing purposes) or hierarchical (showing parent-child relationships). They can also be binary where they connect just two items, or n-ary where any number of items can be connected. Many hypermedia systems also offer typed links. They indicate the specific nature of the relationship shared between nodes rather than just obliquely stating that some unknown form of relationship exists.

2.4. Hypermedia Structure

Structure refers to inter-data relationships [Nürnberg et al. 1996]. Hence hypermedia structure is what forms the relationships between nodes. Any element that contributes to the formation of the relationships between nodes can be considered hypermedia structure. Hypertext links, anchors and the endpoints of links are all examples of hypermedia structure. This thesis does not consider nodes to be hypermedia structure as they represent the data that is being connected by hypermedia structure.

Figure 2.1 uses the example hypertext link of Figure 1.1 to show how it is possible for hypermedia structure to exist independently of the node resources they are being used to connect. Figure 1.1 shows two document node resources connected by hypertext link structure. The two documents, 'Radio' and 'Music', are examples of node resources. The hypermedia objects (e.g. Anchor A1 and Endpoint E1) and the connections between hypermedia objects (e.g. the black line between Anchor A1 and Endpoint E1) are examples of hypermedia structure.

As will be explained in Section 2.12 hypertext links (through the concept of open hypermedia) can be stored as independent entities within link databases. Figure 2.1 shows such an example where the hypertext link of Figure 1.1 has been broken down into 7 separate hypermedia objects and they have been stored independently (of nodes) within a link database. Each hypermedia object is an example of hypermedia structure. The documents (examples of node resources) have also been stored separately within a document repository.

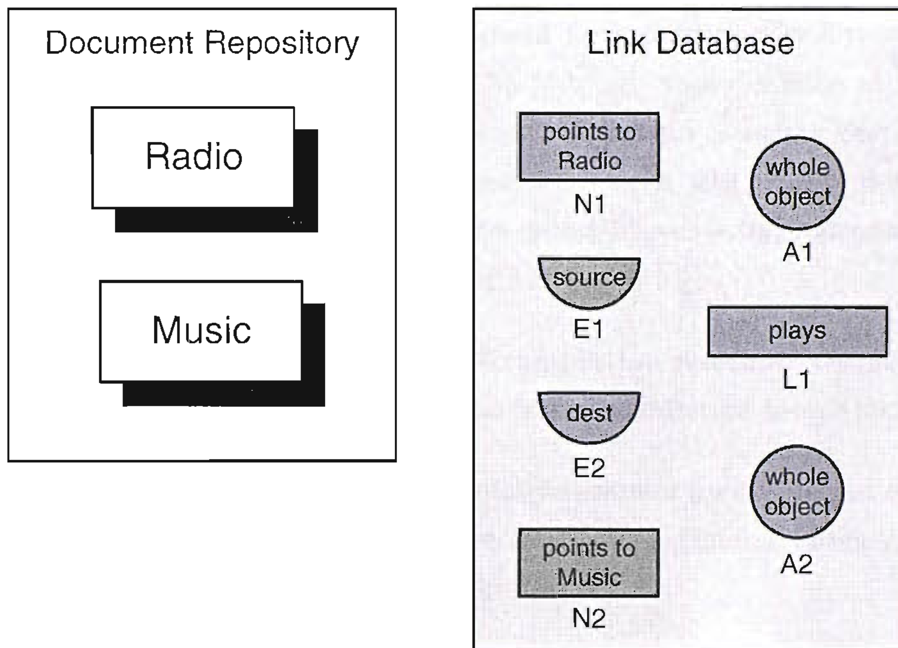


Figure 2.1: Documents and hypermedia structure stored separately and independently from one another. Documents are stored in a document repository and hypermedia structure is stored within a link database.

Hypermedia structure is important to this thesis, since the role of the OCS data model is to define how hypermedia objects may be organised when being connected together to build hypermedia structure (Chapter 7 onwards).

2.5. Re-use in Hypermedia

Re-use also plays an important role in the OCS data model as it is the re-use of hypermedia objects and connections between objects that enables the opening up of hypermedia structure (Computer Science Contribution 1).

As a general concept, re-use is also a recurring theme throughout hypermedia and hypermedia system evolution:

- *Reference Links*. When a source document uses a hypertext link to reference another document, that destination document is re-used at its original location. This prevents wasteful duplication of the destination document's contents being pasted as part of the source document's content. Furthermore it allows the destination document to be referenced multiple times (i.e. re-used) by several source documents.

- *Xanadu Transclusions*. These are a specialised form of hypertext link within the Xanadu hypermedia system (Section 2.7.3). Through a combination of recording document content at one permanent location only and ensuring that only one copy of a document's content exists, transclusion links help prevent the wasteful duplication of re-used document content within a hypermedia environment. This is explained in greater detail in Section 2.7.3.1.

- *Virtual Documents*. These are documents compiled at run-time comprised of re-used document segments possibly originating from different source locations.

- *Composites*. Composites act as a representative element for a group of individual nodes and links. This is an example of re-use as the individual entities are being re-used (via references) within the composite.

- *Open Hypermedia Links*. Because open hypermedia links are stored separately from document content the links themselves can be re-used to point to multiple source or destination documents. Microcosm's generic links are a good example of open hypermedia links (Section 2.12.3) [Davis et al. 1992].

- *OHS Architecture*. A prominent feature of Open Hypermedia Systems is that they enable integration with (i.e. re-use of) third party client applications (Section 2.12).

- *CB-OHS Architecture*. This whole architecture is centred round component re-use. The idea is that all the components of one layer (of a CB-OHS) should be compatible to be re-used with all the components of another layer of the CB-OHS (Sections 2.14 and 2.15).

2.6. Elementary Composition of a Hypermedia System

The essential elements of a hypermedia system are the client, link service and storage component. The evolution of these components within hypermedia systems is shown in Figure 2.2. The client is used for both viewing and editing the content and hypermedia structure of electronic documents. The link service is used to provide the hypermedia functionality for the client so that the client can create and navigate hypertext links. The storage component is used to store hypermedia structure. Depending on the type of the hypermedia system influences whether the document content is stored by the client or in the storage component.

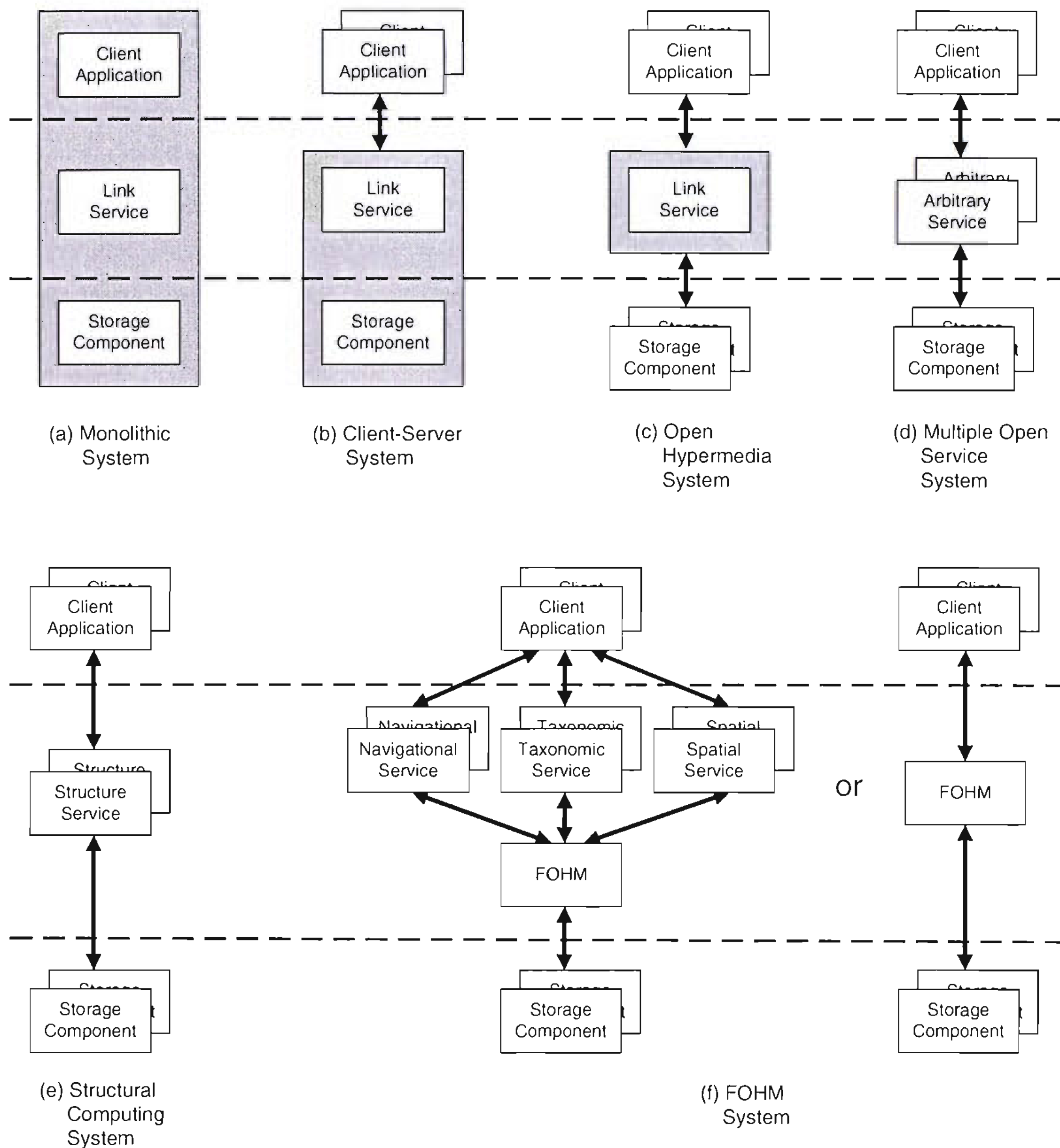


Figure 2.2: Evolution of hypermedia systems.

2.7. The Early Years

Three pioneers essentially introduced the world to the concept of hypertext. They were Vannevar Bush, Douglas Engelbart and Ted Nelson.

2.7.1. Bush

Vannevar Bush is credited as being the grandfather of hypertext. He realised that even in 1945 all the world's information could not be reasonably managed without some form of mechanistic aid for the user. In his seminal paper "As We May Think" Bush advanced a new method for managing information called the memex (memory extender) [Bush 1945]. Its role was to store a library of information, e.g. books, records and communications, all tied together via associative links.

The basic premise behind the memex was that "any item may be called at will to select immediately and automatically another". Thus it was Bush who introduced the associative link as the foremost structure present within hypermedia. Even in 1945 Bush seemed to accurately foresee the general processes involved when forging a (hypertext) link between information items:

> When the user is building a trail, he names it, inserts the name in his code book, and taps it out on his keyboard. Before him are the two items to be joined, projected onto adjacent viewing positions. At the bottom of each there are a number of blank code spaces, and a pointer is set to indicate one of these on each item. The user taps a single key, and the items are permanently joined...
>
> Thereafter, at any time, when one of these items is in view, the other can be instantly recalled merely by tapping a button below the corresponding code space.
>
> *V. Bush, "As We May Think"*

Bush reasoned that tying two information items together was akin to how the human mind works as it too organises information by association.

Bush envisioned the memex to be built within a conventional desk. Figure 2.3 shows what a memex may have looked like. On top lies a scanning mechanism for photographing material into memex storage which would be stored on microfilm. The microfilm would subsequently be viewed on the two translucent screens also atop the desk. The inside of the desk would contain the internal workings of the memex mechanism itself along with the microfilm storage area.

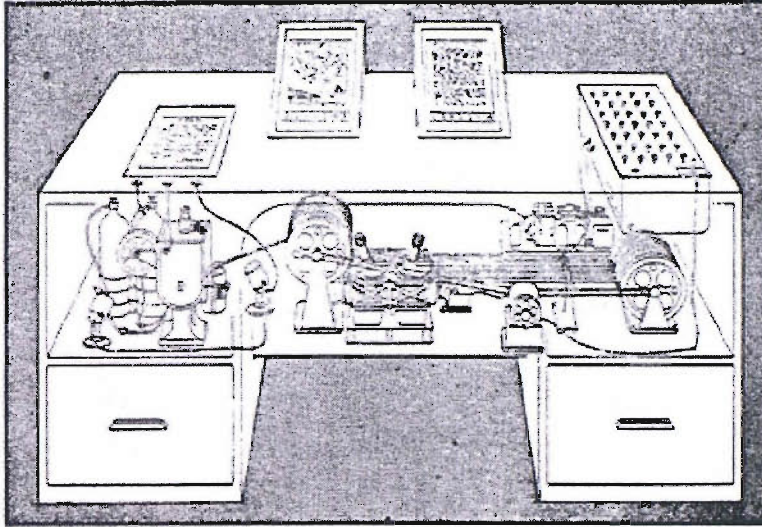


Figure 2.3: What the memex may have looked like [Crimi 1945].

Due to the technological limitations of the 1940s the memex was never realised.

2.7.2. Engelbart

Some twenty years later another visionary, Douglas Engelbart, took up the hypertext cause. He envisaged hypermedia as a tool to augment human intellect. To this end Engelbart developed the NLS/Augment system [Engelbart and English 1968], an all-encompassing environment designed to be adequate for all a user's works needs, storing work documentation, and supporting planning, debugging and communication (such as e-mail). Engelbart, like Bush, also saw the benefit of links between data items. This was because at the heart of the NLS was the use of hierarchical and non-hierarchical links between segments of text. Such links allowed the general re-use of documents through out the system in order to provide users with greater management and control of information.

2.7.3. Nelson

Ted (Theodore) Nelson is viewed as the father of hypertext. He is the man responsible for coining the term 'hypertext' [Nelson 1981]. Nelson imagined hypertext as the vehicle for extending the ways in which information is utilised and navigated. His life's work has been devoted to developing the Xanadu project, a hypermedia system which places the entire world's literature online [Nelson 1999b].

Re-use plays a key role in Xanadu, typified by Xanadu's approach to document content construction. All Xanadu content data is assigned to permanent addresses

which is archived at what is called a permascroll (a scroll of permanent addresses) [Nelson 1999c]. This is an append-and-read-only sequential repository where content is stored in no particular order. Figure 2.4 shows an example.

Xanadu employs content lists in order to present documents in the correct sequential order. They act as the skeletal structure for loading documents with content data. As indicated by Figure 2.4, content lists contain pointers that referentially point to the text within a permascroll. In this way documents are appropriately structured for presentation.

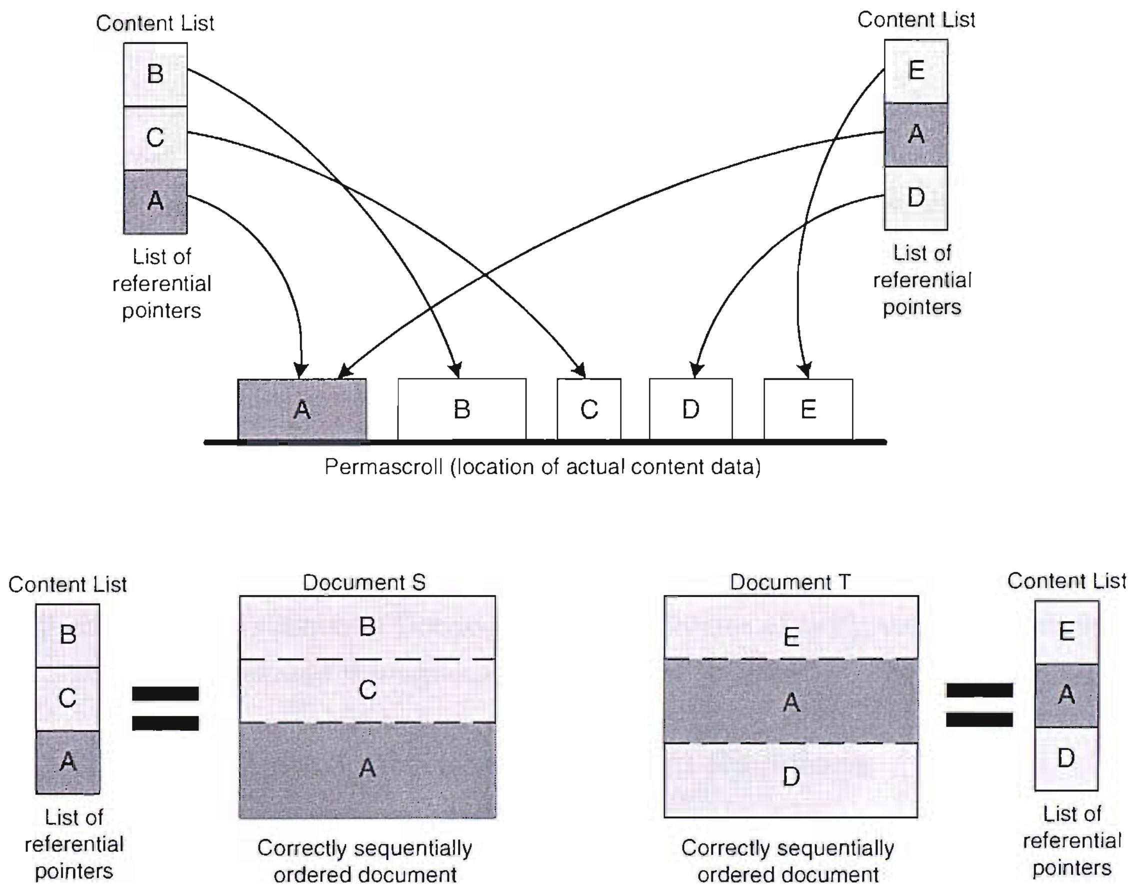


Figure 2.4: Xanadu permascroll scenario: Five pieces of text are archived in a permascroll, but in the wrong sequential order for presentation to the user. To resolve this, two content lists have been employed which point at the permascroll to referentially re-structure the text for final presentation within the appropriate documents.

The net result is that the same Xanadu content data can be re-used to be members of different documents as the same data can be referentially contained within different content lists. This is demonstrated within Figure 2.4 by the content data recorded at permanent location A being included in the content lists for both documents S and T.

Xanadu uses hypermedia structure to highlight the re-use of the same content between different documents. This takes the form of transclusion links. They are links between documents whose content lists point to the same permanently addressed text [Nelson 1995]. An example of a transclusion is shown within Figure 2.4 where both documents S and T re-use the content data recorded at permanent location A.

There are many scenarios where transclusions may be useful. The following three subsections provide three such instances.

2.7.3.1. Preventing Duplication of Re-used Document Content

This subsection follows on from Section 2.5 in explaining how transclusion links play a significant role within Xanadu to help prevent wasteful duplication of re-used document content. Of importance are Xanadu's constraints of permitting only one copy of a document's content to be recorded and that that content must be stored at one permanent location only. Such restrictions mean that transclusions (document content being used within more than one content list) will always reference the same instance of a given document's re-used content. Thus prevented is the unnecessary duplication of re-used document content within the hypermedia environment.

This contrasts with traditional hypermedia systems, e.g. the World Wide Web. They typically take no interest as regards restricting the number of copies that may exist of a given document's contents. Consequently, the contents of any given document may be wastefully duplicated throughout the hypermedia system.

2.7.3.2. Highlighting Re-used Document Revisions

Another facility offered by transclusion links is that they can be used to identify what contents different document revisions have in common. Figure 2.5 shows an example. Each transclusion link between each document revision indicates where the same content simultaneously exists (i.e. is being re-used) within each document.

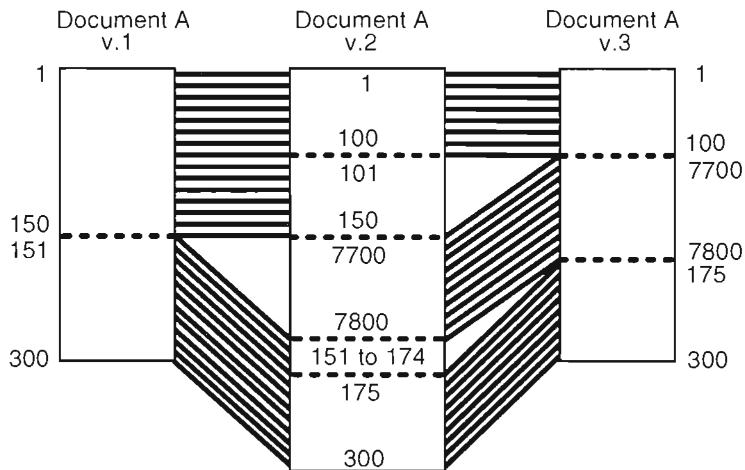


Figure 2.5: Transclusion links between different revisions of the same document.

Such use of transclusion links can benefit tracking the developmental progress of different revisions of a document. For example identifying which document content segments have been brought forward or discarded during a given document's evolution.

2.7.3.3. Clickable Re-used Content Segments

Transclusion links can also be made clickable [Nelson 1999a] in order that a user may click on a transclusion link and move between the different documents that reference the same content segments. This is a useful facility when quoting one document within another. Figure 2.6 shows such an example scenario where Document Y contains a quote from Document X.

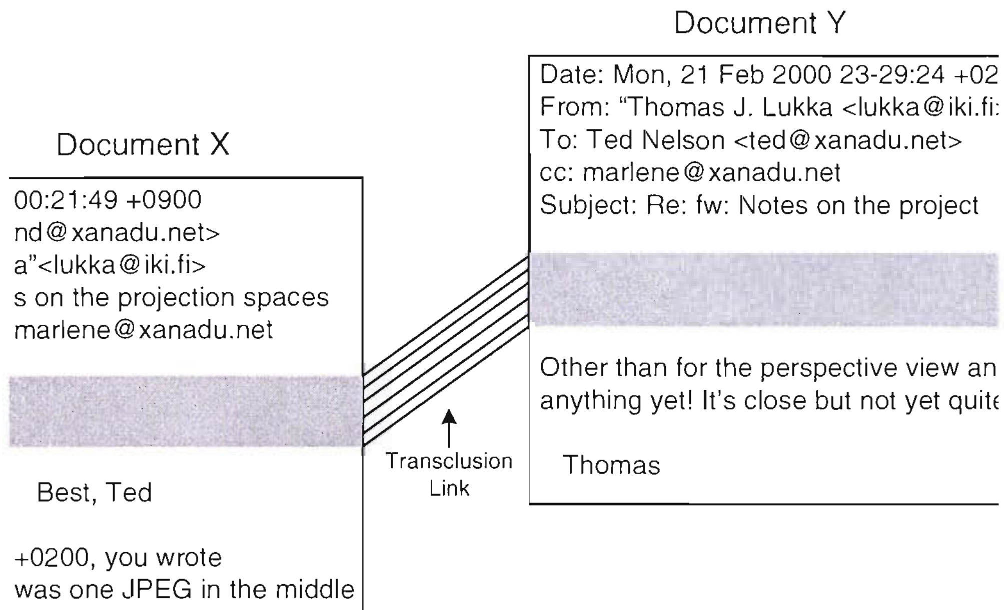


Figure 2.6: Transclusion link showing the re-use of a quote between two documents.

Because the quote in Document Y is an exact reproduction of a segment of Document X's content, then the quote in Y will be created by pointing at the same permanent address as the quote source in Document X. Figure 2.7 shows this in terms of a Xanadu permascroll scenario. The contents of the quote are recorded at permanent location B, hence both documents' content lists point at permanent location B.

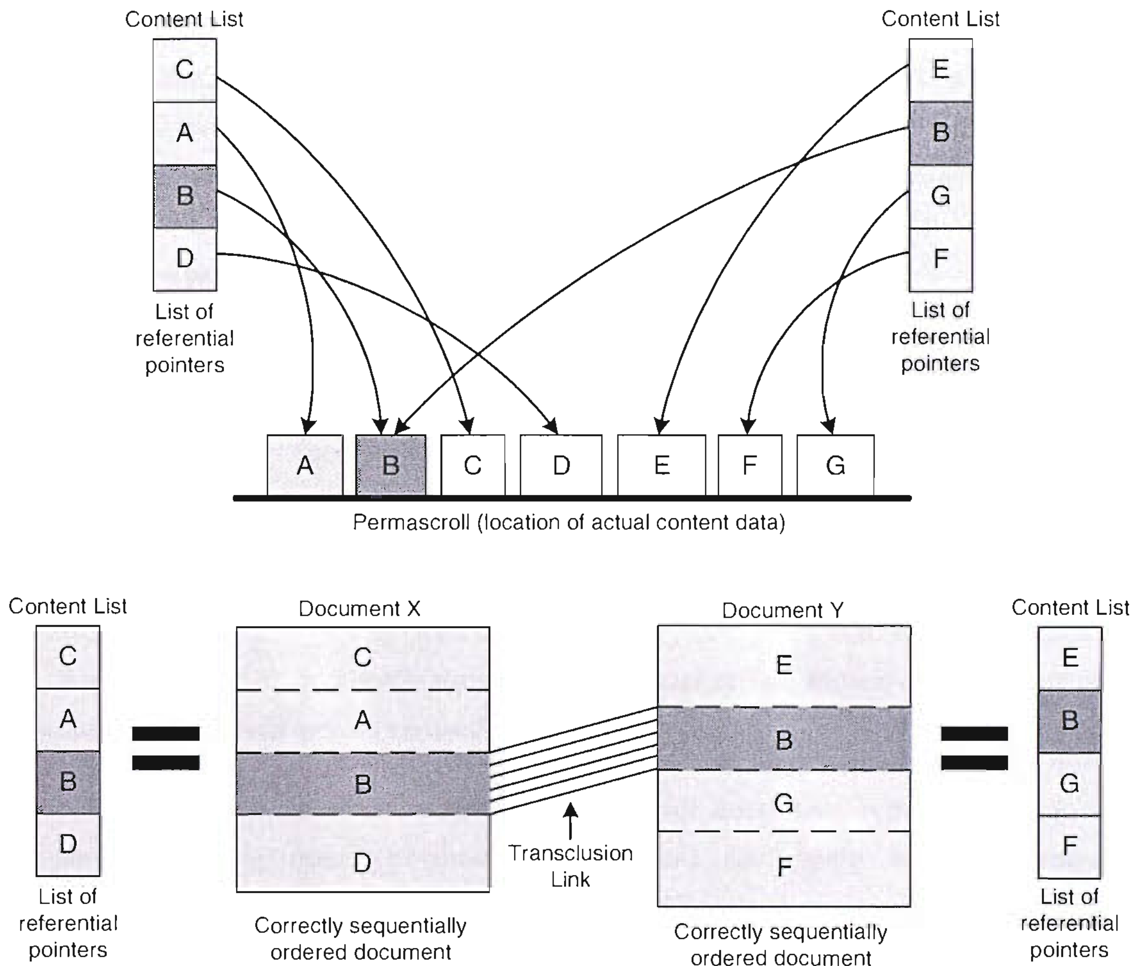


Figure 2.7: Permascroll scenario depicting the transclusion link of Figure 2.6.

By making transclusions clickable, a user who reads the quote can now click on the quote (marked as a transclusion link) and be taken to the original source of the quote to learn more about the quote in its original context.

2.8. Monolithic Systems

Monolithic hypermedia systems represent the earliest hypermedia systems to be implemented. Examples include NLS/Augment [Engelbart and English 1968], ZOG [McCracken and Acksyn 1984], Neptune [Delisle and Schwarz 1986], Intermedia [Yankelovich et al. 1988], NoteCards [Halasz 1988], KMS [Acksyn et al. 1988] and HAM [Campbell and Goodman 1988].

Monolithic hypermedia systems are stand-alone closed systems. They capture all hypermedia system elements within the one application (Figure 2.2(a)). The problem

with this arrangement is that they were designed to provide the entire environment within which users could create and view hypermedia material. Hence they were closed from the environment in which they were operating. Their limitations have been well documented by Meyrowitz [Meyrowitz 1989]. What is evident is that a central reason for their lack of success was because they failed to embrace the concept of re-use:

- *Inability to re-use legacy applications.* Because neither communication protocols nor application programmer interfaces (APIs) were published, users would be forced to forego their favourite browsing and editing tools if wanting to adopt hypermedia functionality.

- *Support proprietary document types only.* To re-use existing documents created outside the monolithic system meant that they had to be converted to a suitable format. This requires both time and additional effort. Moreover the converted documents could not be re-used outside the system.

- *Support a fixed set of data formats only.* Not all document types or documents stored outside the hypermedia store could participate with hypermedia functionality as only the documents stored within the hypermedia store could be linked to.

- *Embedded mark-up.* Hypermedia mark-up and hypertext links would be inserted directly into documents. This makes changing the basic link model in the future difficult as the mark-up in every document would need updating.

This lack of re-use capability meant that users were reluctant to adopt hypermedia technology. Moreover monolithic hypermedia systems represented isolated islands of information [Wiil 1998] as existing applications could not be re-used with monolithic systems and they did not allow the applications of one hypermedia system to communicate with another hypermedia system.

2.9. Client-Server Hypermedia Systems

In the late 1980s the hypermedia community set about investigating the concept of open hypermedia. This led to the abstraction of the client application element from the hypermedia system as shown by Figure 2.2(b). This was to enable hypermedia systems to re-use third party applications. These systems became known as Client-Server Hypermedia Systems [Nürnberg et al. 1997]. A crucial aspect that made them

different to Monolithic Systems was that they specified and published the interfaces that applications needed to adopt to enable them to be re-used by the hypermedia system. Two common types of client-server hypermedia systems evolved [Wiil 1999]:

- *Link Server Systems (LSSs).* They manage hypertext links separate from document content using a link protocol to dynamically apply links to document content at runtime. Sun's Link Service [Pearl 1989] serves as an example.

- *Hyperbase Management Systems (HBMSs).* Otherwise identical to LSSs, a key difference is that they manage both the hypertext link and node data within the same system. Examples include the Danish HyperBase [Wiil 1993] and the World Wide Web (Section 2.10).

Client-Server Hypermedia Systems were ultimately rejected as being truly open systems. The problem was that Client-Server Systems embed hypermedia structure within document content. This makes hypermedia structure become part of document content. Thus only those applications that understand this "document content + structure content" hybrid can participate with the hypermedia system. Hence, because both the hypermedia structure and document content format are fixed meant that third party applications cannot be re-used within Client-Server Hypermedia Systems.

The problem of embedding is a theme that will be revisited within the thesis. This is because the main problem encountered by the OCS data model when attempting to re-define the way structure is organised is the embedding of object references in the hypermedia objects that comprise hypermedia structure.

2.10. The World Wide Web

The World Wide Web (WWW) [Berners-Lee et al. 1992] is an example of a Client-Server HBMS as it stores document content and hypermedia structure together. It is also the only hypermedia system in widespread use today. This is largely due to its standardized protocols that are relatively simple to implement [Cailliau and Ashman 1999]. These are the HTML language, which provides standardized hypermedia services, and the HTTP protocol, which provides standardized access between Web servers and browsers. But it is also these same two protocols that also make the WWW a closed hypermedia system.

First, the HTML format prevents the re-use of an open set of applications within the Web. This is because both document content and hypermedia structure are melded together to make a proprietary document format. Thus the Web is bound to only those applications that understand this document format. And second, the Web restricts communication to be via HTTP. Other applications that communicate using different protocols (e.g. CORBA [OMG 1998] or DDE [Microsoft 2004]) cannot be re-used within the Web.

Of particular relevance is that the hypertext links between HTML documents are embedded in the source documents. This is one of the major criticisms aimed at the World Wide Web, because it prevents efficient re-use of hypermedia documents. The problem being that a Web document cannot be re-used without also being forced to re-use the other documents referenced by the embedded links.

Thus far all these restrictions have not impeded growth of the World Wide Web. But in the long term these constraints may prove to be both inadequate and frustrating for the majority of users as converting all electronic documents to HTML will be a time consuming and costly exercise. Moreover users may find themselves at a greater disadvantage after integrating their existing documents with the Web. For example typical Web browsers do not allow editing bitmap images. Thus users will lose this facility when using the Web to access bitmaps.

Moreover, the World Wide Web Community (W3C) do not themselves regard the Web as a finalised system. Evidence of this is the Web community's current focus on developing the Semantic Web [Berners-Lee 1998]. It provides machine-readable descriptions that add further meaning to, or replace the content of Web documents. The intention is to enable such descriptions to be shared and processed by automated tools in order to improve resource discovery via the WWW. The idea is that machine-readable descriptions should enable a more efficient evaluation of the data present within a Web page compared with encoding data using non-standardised descriptions in HTML.

However, if the World Wide Web is to be the vehicle for achieving the hypermedia community's goal of establishing hypermedia as the central paradigm within the computing environment, then the Web will need to develop into a more open system.

2.11. The Dexter Hypertext Reference Model

Also in the late 1980s the hypermedia community became increasingly frustrated that hypermedia systems were still not interoperable with one another. This was in respect of different hypermedia systems still being unable to share the same hypermedia structure (e.g. links) or even data (e.g. documents). Therefore the Dexter Hypertext Reference Model [Halasz and Schwarz 1990; Halasz and Schwarz 1994] was devised. This was to engender discussion about future interoperable hypermedia system standards. It acted as a data and process model against which future hypermedia models could be compared. And it defined the terminology and semantics of basic hypermedia concepts, particularly in respect of links and anchors.

The Dexter Model breaks a hypermedia system down into three layers (Figure 2.8). The storage layer is composed of a hierarchy of data-containing components (e.g. documents) inter-connected by relational links. Together they comprise the hypertext network. The within-component layer is responsible for the contents and structure within components. But the Dexter Model does not elaborate upon this layer as it is considered to be outside the scope of the hypertext model. The role of the run-time layer is to realise the functionality of users being able to access, view and manipulate the hypertext network structure.

The Dexter Model also employs two interfaces. The first is the anchoring interface. Anchors can specify a location, region, item or substructure within a component such that they do not depend on any knowledge about the internal structure of a component. The second Dexter Model interface is the presentation specification. It describes how components are presented to the user, for example whether a component should be opened for viewing or editing.

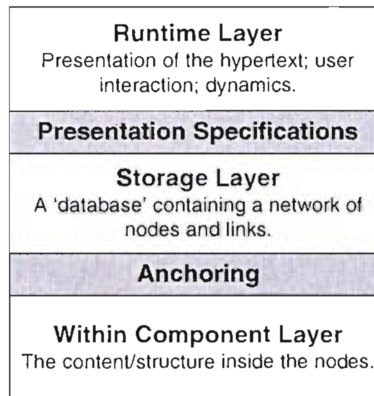


Figure 2.8: Dexter Hypertext Reference Model (adapted from [Grønbæk and Trigg 1994]).

However the shortcomings of the Dexter Model have been well commented [Malcolm et al. 1991; Grønbæk and Trigg 1994; Leistøl 1994]. For example, like Client-Server Hypermedia Systems, the Dexter Model did not adequately address the issues for third party application re-use. This is because it did not envision third party applications to have control over document content. This is evidenced by the model failing to distinguish between components whose contents are managed by the hypermedia system and contents managed by third party applications. Therefore the Dexter Model was rejected as a complete solution for hypermedia systems. But this should not diminish the Dexter Model's importance as it proved to be the stepping stone for advancing towards more open systems, such as Open Hypermedia Systems which are discussed next.

2.12. Open Hypermedia Systems

The early 1990s onwards saw the development of Open Hypermedia Systems (OHSs). They were classified as open since they allow third-party applications to be re-used. This is possible since no hypermedia structure is embedded in document content, including hypertext links being embedded in source documents or anchor data being embedded in destination documents. Instead hypertext links (which include anchor data) are stored as independent entities in link databases. Thus at display time, links are superimposed on document content along with the relevant data for the dynamic calculation of anchors to enable them to be positioned within document content.

The result was that hypermedia systems could link to any document of any content format, and individual hypermedia documents could be re-used to produce different

views of a hypertext network. Both these scenarios were not possible with closed hypermedia systems, such as the World Wide Web (Section 2.10).

Furthermore, applications do not have to subscribe to all the services offered by an OHS. They can pick and choose which specific services they want to use. For example clients can continue to use (i.e. re-use) their existing file system instead of being restricted to a hypermedia system's storage service. The outcome was the abstraction of the storage element of a hypermedia system as shown by Figure 2.2(c).

2.12.1. Similarities between Different Open Hypermedia Systems

Examples of Open Hypermedia Systems include Microcosm [Davis et al. 1992], SP3 [Leggett and Schnase 1994], DHM [Grønbæk et al. 1994], Chimera [Anderson et al. 1994] and HyperDisco [Wiil and Leggett 1996]. These systems have much in common. They all provide basic associative hypermedia structuring services: the ability to create, delete and modify links and anchors, and the ability to traverse links [Wiil and Nürnberg 1999]. They all also superimpose hypermedia structure on document content at display time. The reduction of an OHS to a solitary link service component has also meant that most OHSs behave as a middleware service [Bernstein 1996] – they not only meet the requirements of a wide variety of applications, but they run on multiple platforms, are distributed, publish communication protocols and APIs.

2.12.2. Microcosm

A prime example of an Open Hypermedia System is Microcosm [Carr et al. 1994]. It functions as a set of autonomous communicating processes designed to supplement the operating system [Davis et al. 1992]. The Microcosm model is shown in Figure 2.9.

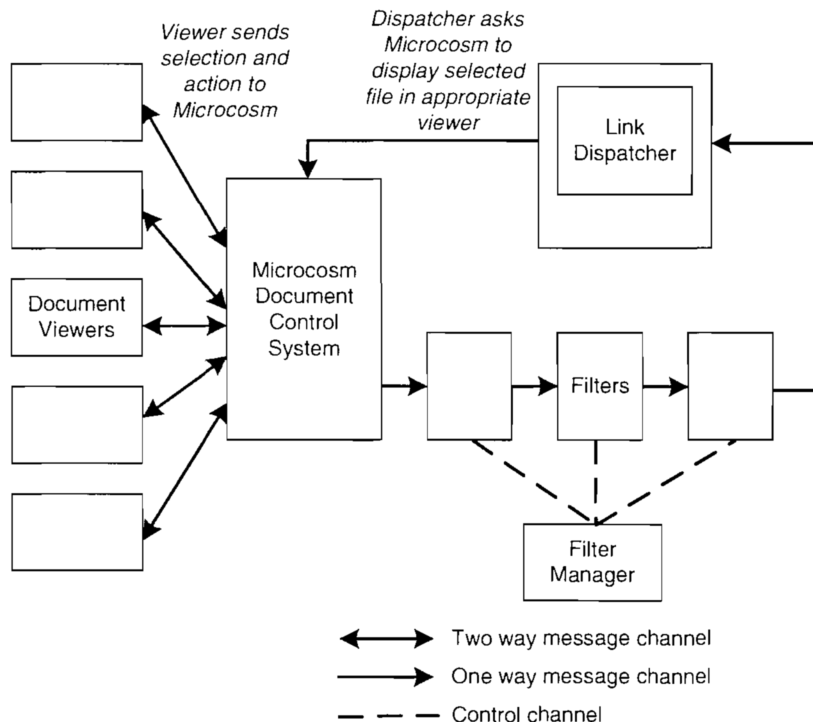


Figure 2.9: Microcosm architecture.

Users interact with a viewer (any application that can display data) in order to send messages to Microcosm. Microcosm then dispatches the same messages through a chain of filters. Each filter can either block the message, pass it on, or change the message prior to passing it on. Dependent on the message contents, some filters may even add new messages to the chain.

Having passed through the filter chain, Microcosm messages will arrive at the Link Dispatcher. Its role is to examine each message for actions to carry out (such as links to follow).

Linkbases (link databases) are particularly important Microcosm filters. They store all linking information separately from the documents they reference. This means no embedded mark-up within document data is necessary. Hence all data is accessible to, and editable by, the application that created it.

When a link request message arrives at a linkbase, the linkbase looks up the source of the link (stored either in a database or data file) in order to identify details of the destination of the link. This information is then forwarded to the Link Dispatcher, which then asks Microcosm to display the selected file in the appropriate viewer.

Other Microcosm filters include processes to aid navigation (such as the History Mechanism) and processes to compute dynamic links. Implementing hypermedia functionality as filters means that the behaviour of the system can easily be varied without having to change the existing components of the system. This is possible through dynamically installing new filters or dynamically removing or re-ordering existing filters.

2.12.3. Re-use in Open Hypermedia Systems

The hypermedia functionality offered by Open Hypermedia Systems has benefited enormously from the concept of re-use. Microcosm is one such example. Its ability to store links separately in link databases has meant that hypertext links can be re-used in their entirety [Davis et al. 1992]. Two Microcosm link types demonstrate this:

- *Local Link*. This is a link from a particular object at any point in a specific source document that connects to a particular object in a destination document. Thus the same hypertext link can be re-used multiple times within the same source document.

- *Generic Link*. This is a link from a particular object at any position in any source document that connects to a particular object in a destination document. This enables the same hypertext link to be re-used multiple times within different source documents.

2.12.4. A Lack of Consensus

Solving the third party application re-use problem meant that users could continue to use their favourite applications which could now be emblazoned with hypermedia functionality. This added further incentive to adopt the hypermedia paradigm when organising information. However, the hypermedia community next had to agree on which properties of a hypermedia system should be opened in order to make hypermedia functionality more readily available to all applications and services in the computing environment. This resulted in a lack of consensus [Wiil and Leggett 1992]. All OHSs offer the basic hypermedia structuring services, but different OHS designers also identified additional properties that ought to be opened. For example Davis [Davis 1995a] lists six properties that should be opened in order to classify a hypermedia system as truly open:

1. *Size.* Be able to import new nodes, links, anchors and other hypermedia objects without limitations to the size or the number of objects that the system may contain.

2. *Data Formats.* Allow the use and import of any data format, including temporal media.

3. *Applications.* Allow any application access to the link service enabling participation in hypermedia functionality.

4. *Hypermedia Data Models.* The hypermedia data model should be extensible and configurable so that new data models can be incorporated.

5. *Platforms.* Be able to implement the system on multiple distributed platforms.

6. *Users.* Support multiple users, allowing each individual to maintain their own private view of objects in the system.

Due to these contrasting opinions, the characteristics of different OHSs have often varied. For example:

- *Hypermedia Data Models.* Different Open Hypermedia Systems often implement different data models, e.g. the number of endpoints of a link or whether they have a notion of anchors.

- *Hypermedia Services.* Some Open Hypermedia Systems support just basic hypermedia services allowing links and anchors to be attached to existing data, whilst other systems offer more advanced features such as support for collaborative authoring of hypermedia structures and contents.

- *Hypermedia Structures.* All Open Hypermedia Systems support hypermedia linking structures. However some systems support other types of structure as well, e.g. composites, information retrieval links, spatial structures, taxonomic structures, etc.

Such different characteristics tended to result in incompatibility between different OHSs, e.g. a user being unable to follow a link from one OHS to another. The result was that each OHS would operate in an isolated and limited manner instead of being part of a larger unified hypermedia environment.

2.13. The Open Hypermedia Protocol Project

The Open Hypermedia Protocol (abbreviated to OHP) was the open hypermedia community's answer to solve the incompatibility between different OHSs [Reich et al. 1999a]. It is described in further detail in Chapter 3 since it played a significant role in the conception of the OCS data model. In the meanwhile it is sufficient to say that the OHP has provided a standardised architectural framework for Open Hypermedia System components, a Core Data Model for representing hypermedia structure and a range of standardised Content Handler Interface protocols. Notable among these protocols is the OHP-Nav protocol. It is a standardised linking protocol that, although not yet complete, demonstrates how it is possible to share hypermedia structure between different hypermedia systems. In this way inter-operation between different OHSs is enabled. A positive outcome from the OHP is that it makes it easier to program third party applications to communicate with a wide range of OHSs thereby making hypermedia functionality more readily available within the computing environment.

2.14. Multiple Open Systems

Following the development of Open Hypermedia Systems was the Multiple Open Systems (MOS) research thread [Wiil et al. 2001]. Its premise is re-use at all levels of the OHS architecture. It builds upon the representation of an OHS as a middleware service[2]. The intention being to produce a highly flexible architectural framework where each layer of the OHS architecture is defined as a separate service layer. The different layers of a Multiple Open System (Figure 2.2(d)) are defined as:

- *Application (Services) Layer*. This corresponds to the OHS interpretation of an open set of applications.
- *Middleware Services Layer*. This is shown as a set of arbitrary services in Figure 2.2(d). It corresponds to the Link Service layer, but is significantly expanded to provide arbitrary services and not just navigational hypermedia structuring facilities. This can include collaboration services, integration and interoperability services as well as hypermedia structure services.

---

[2] Open Hypermedia Systems' middleware service is encapsulated by a linkserver component whose service is hypermedia navigation functionality.

- *Foundation Services Layer*. This corresponds to the storage layer of an OHS. Again it is much expanded, and can include concurrency control services, versioning control services, as well as hypermedia structure storage services.

Each service at each layer is also split (modularised) into separate components where each component provides some well-defined functionality using a well-defined interface. Such modularisation enables re-use between all layers within an OHS architecture. For example a single application can use multiple services; a single middleware service can be used by multiple applications; a single middleware service can use multiple foundation services; and a single foundation service can be used by multiple middleware services. The result is that Multiple Open Systems are often referred to as Component-Based Open Hypermedia Systems (CB-OHSs). This is because a MOS breaks an OHS down into multiple arbitrary components.

Construct [Wiil 2001] (Section 2.15.1) is an example of a Multiple Open System. It provides an environment to assist with the development of different hypermedia structuring services and collaboration services.

2.15. Structural Computing

Structural Computing [Nürnberg et al. 1996; Nürnberg 1999] is a specialisation of Multiple Open Systems research. This is because it adopts the same three-layered framework as Multiple Open Systems with the exception that instead of an arbitrary Middleware Services Layer, a Structural Computing System employs a Structure Services Layer. The difference with the Structure Services Layer is that it is restricted to providing services that define operations over structure only.

Figure 2.2(e) shows an example of the composition of elements within a Structural Computing system. Because Structural Computing systems adopt the same three-layered architecture as MOS research, they too are categorised as CB-OHSs.

Fundamental to Structural Computing is the basic unit of structure called the structural atom. Structural atoms are based upon the notion of 'relationship', i.e. a structural atom is used to relate items together. They are also the elementary building block for constructing everything within the computing environment. This includes data items (e.g. a computer file would be modelled as structure with content), and high-level structural abstractions. In this case structural atoms are related to one another in order to build hypertext links, taxonomic links and spatial

associations. The importance of structure within Structural Computing is such that Structural Computing elevates structure to be a first class entity and asserts the primacy of structure over data within the computing environment.

The result is that Structural Computing classifies hypermedia as a subset of Structural Computing. Advocates of Structural Computing hold the view that the notion of hypermedia linking is too closely allied with the practice of link navigation. This is because using structure to link information items together solely for the purposes of navigation restricts the potential applications for structure. Their reasoning is that structure can also be used to address different problems in multiple domains [Wiil 1999]. Examples of those other domains include hypermedia literature [Bernstein et al. 1991], argumentation support [McCall et al. 1990], information analysis [Marshall and Shipman 1995], hypermedia art [Leistøl 1994; Sawhney et al. 1996], and taxonomic work [Nürnberg et al. 1996]. Figure 2.10 shows a conceptual overview of open service provision geared towards middleware hypermedia structural services.

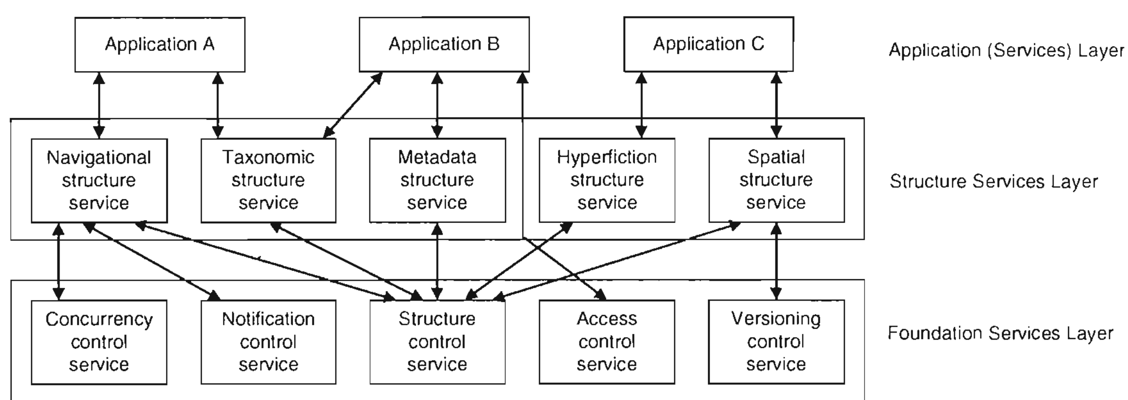


Figure 2.10: Conceptual overview of Structural Computing hypermedia structure service provision (adapted from [Wiil et al. 2001]).

As with MOS, the focus is on re-use once again. For example, Applications A, B and C are clients of different combinations of structure services. The five structure services, in turn, are clients of different combinations of foundation services.

In terms of hypermedia, the attention of Structural Computing has principally concentrated on the navigational, spatial and taxonomic domains. The primary research vehicles have been HOSS [Nürnberg et al. 1996] and the Construct System [Wiil et al. 2000; Wiil 2001].

2.15.1. Construct

Construct [Reich et al. 1999a] is a Component-Based Open Hypermedia System developed at Åarhus and Aalborg Universities in Denmark. It builds on the HOSS [Nürnberg et al. 1996], HyperDisco [Wiil and Leggett 1997] and DHM [Grønbæk et al. 1994] Open Hypermedia Systems. Construct is also an example of a CB-OHS that implements the OHP architectural framework including some of its protocols, e.g. OHP-Nav (OHP and OHP-Nav is explained in greater detail in Chapter 3.)

The Construct architecture, as is the case with all Structural Computing systems, is composed of three layers. An implementation of the architecture is shown in Figure 2.11.

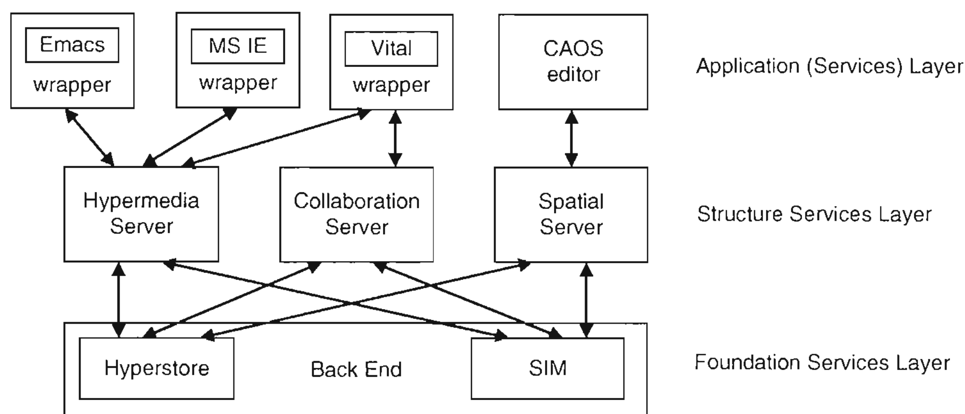


Figure 2.11: Construct CB-OHS architecture.

The Foundation Services Layer is comprised of two types of component. The first is the Hypermedia Store (Hyperstore). It handles persistent storage of hypermedia structure and implements core collaboration services (e.g. transaction, concurrency control, notification control, access control and version control). The second is the Service Information Manager (SIM). It provides naming and location services which enables components to operate in a distributed setting, e.g. over the Internet.

The Structure Services Layer contains three servers. The four example client applications (at the Application Services Layer) utilise each server to offer a particular type of functionality. Emacs and Microsoft Internet Explorer offer clients navigational hypermedia functionality via the Hypermedia Server, Vital offers both collaboration and hypermedia functionality via the Collaboration Server and Hypermedia Server respectively, and the CAOS Editor offers spatial functionality via the Spatial Server (spatial hypermedia is explained in detail in Section 3.3.2.2).

The remainder of this section explains how Emacs communicates with the Construct Hypermedia Server, however the other three applications (MS IE, Vital and the CAOS Editor) communicate with their respective servers in essentially the same manner as described for Emacs.

Emacs is a widely used text editor available on both Unix and MS Windows. It has been extended with Construct navigational hypermedia services through adding a new menu to the Emacs interface with five navigational menu items. These are Traverse Link, Start Link, Create Anchor, Delete Anchor and End Link. The Construct Emacs application is also an example of an application that uses the OHP-Nav protocol (Section 3.2.3) to enable communication between it and the Hypermedia Server. This has been achieved by encapsulating Emacs within a wrapper written in Java. Its role is to mediate interaction between an application and a middleware server. Hence the wrapper translates requests from Emacs into the OHP-Nav protocol and sends them to the Hypermedia Server, and vice versa.

Construct is also mentioned again in the thesis (Section 3.2.8) as it played a significant role in demonstrating true interoperability between OHSs for the first time by implementing the OHP-Nav Protocol.

2.16. The Fundamental Open Hypermedia Model

The Fundamental Open Hypermedia Model (FOHM) [Millard 2000a] is a research tool for investigating inter-domain interoperability. This is the capability of translating the hypermedia structural abstractions of one domain into the hypermedia structures of another hypermedia domain. The basis of the research is to develop a common storage layer that is sufficiently generic to enable the storage of structures of any hypermedia domain. So far interoperability within the navigational, taxonomic and spatial domains have been investigated. Figure 2.2(f) shows the composition of elements within a CB-OHS adapted to FOHM.

The promise of the FOHM architecture is that if all middleware services understand the same fundamental structure abstractions, then the interfaces between all middleware service components and the component services of the foundation layer can be standardised. This means greater interoperability between the components of an OHS should be possible.

Re-use also plays a key role within FOHM since one of its aims is to facilitate the re-use of structures between all three hypermedia domains. FOHM is described in more detail in Chapter 3 as it played a critical role in the development of the OCS data model.

2.17. Summary

Hypermedia system development has gone a long way towards addressing the hypermedia community's aim of making hypermedia the central paradigm within the computing environment. Hypermedia systems are no longer the straightforward "one-size-fits-all" monolithic systems. Today their architectural infrastructure is much more complex. They are envisioned as open sets of middleware components built on top of structure-aware backends serving structural abstractions to open sets of clients.

Re-use has also been a helpful partner in promoting the advantages of hypermedia. This has taken many forms, including third party application re-use, hypertext link re-use within OHSs, as well as the re-use of components within CB-OHSs. The latter has enabled users to pick and choose which services are most appropriate to their needs.

However there is (as yet) no widely accepted Open Hypermedia System in the public domain. This is partly because of the World Wide Web. Its shortcomings have not yet been realised to their fullest extent. The result has been that the concept of open hypermedia has not yet caught on with the general public. Another reason is that the development of hypermedia concepts and hypermedia systems have not stood still. They have moved beyond the traditional navigational domain, and research is now focused on how hypermedia functionality can address a wider range of problem domains, including knowledge management, software development, collaborative authoring and digital libraries.

Chapter 3. OHP and FOHM

3.1. Introduction

This chapter provides background about the Open Hypermedia Protocol (OHP) and the Fundamental Open Hypermedia Model (FOHM). Briefly, the OHP is a research project aimed at enabling interoperability between OHSs, whilst FOHM is a research project that investigates hypermedia domain interoperability. Together they provide data models for hypermedia structure.

OHP and FOHM are directly relevant since the role of the OCS data model is to describe how hypermedia structures (such as those depicted by OHP and FOHM) can be built through connecting individual hypermedia objects. Moreover it is precisely due to the shortcomings of *how* OHP and FOHM connect objects together within their respective data models that fuelled development of the OCS approach.

3.2. The Open Hypermedia Protocol

Primary interest in the Open Hypermedia Protocol (OHP) stems from my working with the Open Hypermedia Systems Working Group (OHSWG) [OHSWG 1997]. This is the group responsible for developing the OHP. Their objective has been to facilitate construction of interoperable Open Hypermedia Systems. This led to the OHP research project being devised. Its initial focus was as a standardised protocol to enable communication between open hypermedia clients and open hypermedia servers [Davis et al. 1996]. But it soon became apparent that a single protocol was not sufficient as it was realised that more aspects of Open Hypermedia Systems needed standardisation. Hence the OHP evolved into the wider reaching aim of developing an open framework to enable application developers to construct hypermedia-aware applications [Davis 1996].

3.2.1. Core Architecture

To support interoperability between OHSs, the OHSWG created the Common Reference Architecture (CoReArc) data model. It is a conceptual architecture which provides a standardised model upon which to base the layout of the components of an OHS. It is shown in Figure 3.1.

CoReArc has direct relevance to the OCS since the OCS data model provides the underlying organisation of hypermedia structure that is dispatched via the interfaces and passed between the components.

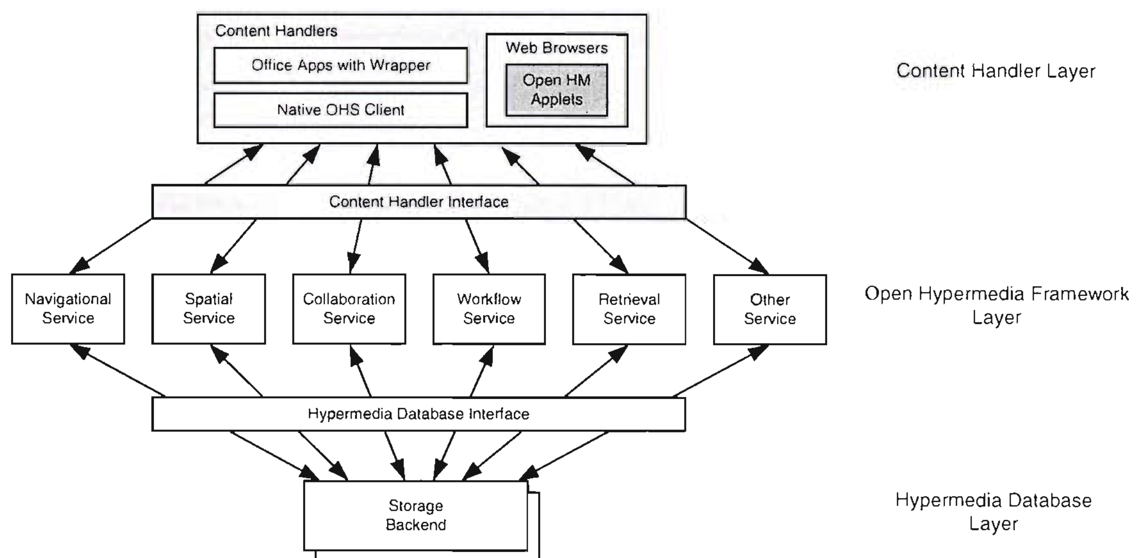


Figure 3.1: The Common Reference Architecture (CoReArc) data model.

CoReArc has been designed to support multiple hypertext domains. These include the navigational, spatial, taxonomic and transclusional hypertext [Nelson 1999a] domains.

As can be seen by Figure 3.1 the CoReArc model identifies three conceptual layers:

- *Content Handler Layer*. Represented by client applications. It handles the storage, retrieval, editing and presentation of contents.

- *Open Hypermedia Framework Layer*. It consists of a set of services providing hypermedia services to the applications of the Content Handler Layer. For example navigation, structuring, integration, collaboration and interoperability.

- *Hypermedia Database Layer.* It provides the storage backend for hypermedia structures.

The conceptual layers are serviced by two main interfaces:

- *Content Handler Interface (CHI).* Enables communication between client applications and hypermedia servers. It is composed of multiple interfaces applicable for different hypermedia application domains. For example one CHI for navigation and another CHI for spatial hypermedia applications.

- *Hypermedia Database Interface (HDBI).* Enables communication between hypermedia middleware services and hypermedia databases. It supports the general storage of hypermedia objects, and provides support for collaboration and distribution in a standardised manner.

CoReArc is a Component-Based Open Hypermedia System (CB-OHS) architecture (Section 2.14). It effectively mirrors the Multiple Open Service System and the Structural Computing System architectures shown in Figures 2.2(d) and 2.2(e). Thus the Content Handle Layer maps onto the Client Application Layer; the Open Hypermedia Framework Layer maps onto the Arbitrary (or Structure) Service Layer; and the Hypermedia Database Layer maps onto the Storage Component Layer.

Referring back to Section 2.11 which discusses the Dexter Hypertext Reference Model. It identified that a major problem with the Dexter Model was that it did not envision third-party applications to have control over document content. This meant the Dexter Model did not enable third-party application re-use. The CoReArc data model improves upon this situation as it explicitly designates that document content should be handled by the Content Handler Layer. Hence third-party application re-use is enabled.

3.2.2. Core Data Model

Also identified by the OHSWG as requiring standardisation are the hypermedia structure data models adopted by Open Hypermedia Systems.

Hypermedia structure data models are the hypermedia objects that make up hypermedia structures and the semantics that describe how those objects are connected together. Standardisation of such models means that an application of one OHS can understand and hence load another OHS's structure. This means each OHS and its applications are not restricted to the structures created by just that one OHS.

For example a standardised navigational data model would allow an application of one OHS to load and follow a hypertext link created by another OHS. This marks a significant step towards achieving OHS interoperability as it prevents OHSs acting in an isolated manner. It was for these reasons that the OHSWG devised the Core Data Model [Reich et al. 1999a] (Figure 3.2).

The OHSWG Core Data Model has direct relevance to the OCS data model since the purpose of the OCS data model is to improve upon the internal organisation of hypermedia structures depicted by such hypermedia data models as the Core Data Model. Moreover, it was through experimentation with and implementation of the navigation classes of the Core Data Model (Section 3.2.3) that led to development of the OCS data model.

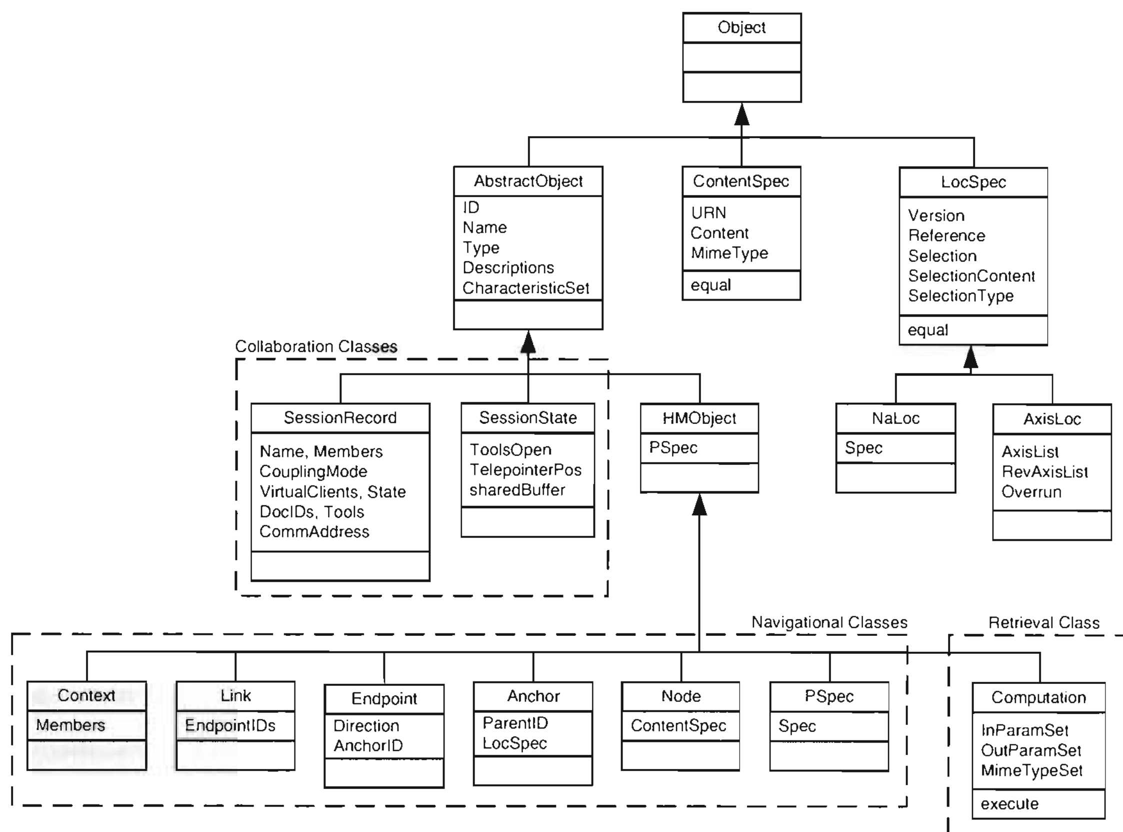


Figure 3.2: The OHSWG Core Data Model.

The Core Data Model is a unified data model that represents a combination of the common concepts found within the hypermedia data models of independently created OHSs. These include the Dexter Hypermedia Model [Halasz and Schwarz 1994], Chimera [Anderson et al. 1994], HOSS [Nürnberg et al. 1997], HyperDisco [Wiil and Leggett 1996] and Microcosm [Davis and Hall 1997].

The result is a common yet extensible hypermedia structure data model. It provides interoperability among hypermedia servers and ensures openness for inclusion of new servers (offering new hypermedia services) in the future. To achieve this the Core Data Model treats the primary characteristics of objects as first class attributes, and the secondary set of characteristics are kept open for future extensions through the use of attribute values.

As can be seen in Figure 3.2 the basic element of the Core Data Model is the abstract class AbstractObject. It consists of an identifier, a name, a set of descriptions, a type, and a set of characteristics. A characteristic consists of a name (String) and a set of values (Strings). All objects below the AbstractObject inherit these first class attributes.

The Core Data Model shows the basic classes and their specialisation into a navigational data model, collaboration data model and computation data model.

The navigational interface is described in more detail in Section 3.2.3 since it was as a result of my developing prototypes based on this interface that led to the OCS data model being created. The other two classes are summarised below.

3.2.2.1. Collaboration Classes

The Collaboration Classes are used to assist collaboration servers and their clients to mediate interaction between multiple "collaboration-aware" clients. As indicated by the Core Data Model the collaboration service is assumed to provide two basic abstractions: Session Record and Session State. The Session Record contains basic information about a collaborative work session, for example its members and their logged-in or logged-out status, the documents currently being shared, and the collaborative mode of sharing (i.e. synchronous or asynchronous). The Session State is used to describe additional information about the session. This can include the tools currently open on shared documents in the session, the current position of a telepointer shared among clients, or a shared buffer used to exchange data between clients.

3.2.2.2. Computation Classes

The Computation Classes enable lightweight clients access to external services that offer advanced functionality. A Computation can be thought of as a black box of functionality [Reich et al. 1999a]. It is defined by its name, a set of input parameter

templates, a set of output parameter templates, and a set of mime types for which the Computation is valid. Clients know the computation function only by its name, which can be invoked and its results are understood by the client even though its workings are completely opaque. In this way a generic client can access complex functionality that would otherwise be unavailable.

For example, a client application may support navigation by finding "similar" images. The notion of similarity may depend on finding characteristics such as the colour space, colour distribution or other meta-data about the image. Clients can offer this functionality to the user by relying on external services that are dynamically discovered and invoked which they only need to call rather than implement themselves. Hence lightweight clients can offer advanced functionality.

3.2.3. OHP-Nav Protocol

The Navigation Classes within the Core Data Model represent the Open Hypermedia Protocol Navigational Interface (abbreviated to OHP-Nav) [Davis et al. 1996; Millard et al. 1998b]. Together these objects form a standardised linking protocol for use in the navigational hypermedia domain.

The OHP-Nav is but one of the Content Handler Interfaces of the CoReArc data model (Figure 3.1). It enables users to traverse hypertext links stored in different OHSs. OHP-Nav acts as the communication medium between Client Applications and the Navigational Service. The latter's function is to endow the client with navigational hypermedia functionality, i.e. the ability to retrieve, create, delete, update and follow hypertext links.

An OHP-Nav hypertext link is composed of the hypermedia objects of the Navigational Class: Context, Link, Endpoint, Anchor, Node and PSpec objects. The associations between the objects are shown in Figure 3.3.

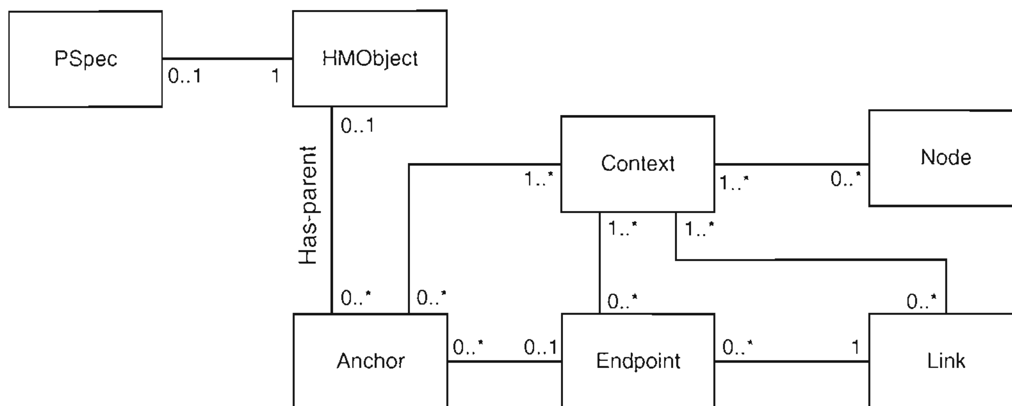


Figure 3.3: Navigation Classes of the OHSWG Core Data Model.

The role of OHP-Nav is to provide a set of standardised services for the creation, modification, retrieval, update and navigation of the following hypermedia objects:

- *HMObject*. This is an abstract superclass that generates no actual instances. All the first class hypermedia objects that contain unique identifiers (Context, Node, Anchor, Endpoint and Link) inherit from HMObject. These objects are typically associated with each other through one-to-one, one-to-many and many-to-many relationships as depicted in Figure 3.3.

- *Context* (first-class object). Contexts are objects implemented as collections of object references such that OHP-Nav hypermedia objects can be added or removed as members of a Context. OHP-Nav Contexts are similar to Hypertexts in Dexter [Halasz and Schwarz 1994] and Contexts in Leggett and Schnase [Leggett and Schnase 1994].

- *Node* (first-class object). A Node[3] is either the resource that is being linked to, or it provides the location of the referenced resource (e.g. a document) via a ContentSpec.

- *ContentSpec* (in-lined within Node objects). A ContentSpec (content specifier) serves as a proxy object for the Node object for the benefit of the hypermedia

[3] The remainder of the thesis distinguishes between OHP-Nav Node objects and the ordinary nodes of a hypermedia system (e.g. documents) by referring to the former as Nodes and the latter as node resource objects or simply as nodes (with a leading lower-case n).

service. An example of a ContentSpec might be a Uniform Resource Name (URN) [Sollins and Masinter 1994].

- *Anchor* (first-class object). An Anchor consists of an identifier and optional location that defines the persistent selection within a Node. The location information is provided by a LocSpec object.

- *LocSpec* (in-lined within Anchor objects). A LocSpec (location specifier) identifies a location in arbitrary content in a client-specific fashion. Examples of LocSpecs include a byte offset and byte extent in a stream of binary data; a pair of (x,y) co-ordinates in an image file; or an entire Node object.

- *Endpoint* (first-class object). Holds the attributes of one end of an OHP-Nav Link object including the traversable direction value of a hypertext link, e.g. source, destination or bi-directional.

- *Link* (first-class object). Represents a relationship between node resources. The role of an individual Link object is to specify the binding together of zero or many Endpoint objects. A Link object also contains a description attribute describing the nature of the relationship shared by the node resources being connected together by the hypertext link of which the Link object is a member. An OHP-Nav Link object should not be confused with a hypertext link.

- *PSpec* (first-class object). A PSpec (presentation specifier) can be associated with any hypermedia object. It stores an opaque specification of attributes specific to a particular application (content handler) that manages the presentation of a hypermedia object at run-time. For example a PSpec associated with an anchor can specify the appearance of that anchor within a document, such as the anchor's colour, style and visibility.

The OHP-Nav protocol is very relevant to the OCS data model and the thesis as a whole for two reasons. First, OHP-Nav hypermedia objects were used as the test ground when developing the OCS data model. Second, Chapters 6 and onwards use OHP-Nav objects to explain and demonstrate the workings and benefits of the OCS data model.

3.2.4. Condensed OHP-Nav Data Model

The OCS data model examples used within the thesis rely on a condensed version of the OHP-Nav data model. These are the absolute minimum objects necessary for

constructing OHP-Nav hypertext links. This makes explaining and demonstrating the workings and the benefits of the OCS data model clearer as there are fewer objects for the reader to concentrate on. Figure 3.4 shows the revised data model.

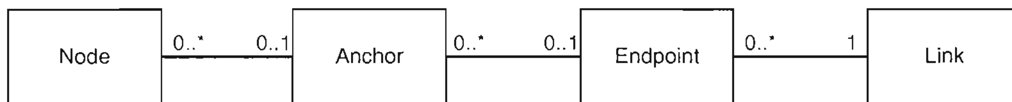


Figure 3.4: Condensed OHP-Nav data model.

The revised data model uses just four inter-connected objects in order to create OHP-Nav hypertext links: Nodes connected to Anchors connected to Endpoints connected to Link objects.

The associations between the four object types are as follows:

- *Node*. Can be associated with zero or one Anchors.
- *Anchor*. Can be associated with zero or one Endpoints; and it can be associated with zero or more Nodes.
- *Endpoint*. Can be associated with zero or more Anchors, but can only be associated with one OHP-Nav Link object as an Endpoint is used to signify the end of just one OHP-Nav Link object.
- *Link*. Can be associated with zero or more Endpoints.

HMObjects have been omitted since they are an abstract superclass and no instances of them are created when constructing OHP-Nav hypertext links. Both PSpecs and Contexts have been omitted since they are not essential for hypertext link construction. Another reason why PSpecs have been omitted is because they have not yet been properly defined by the OHSWG.

Whilst ContentSpec and LocSpec objects are considered part of the condensed OHP-Nav data model, they are not shown in Figure 3.4 because they are not first-class entities. Therefore, due to their in-lined status, ContentSpecs are assumed to automatically be part of a Node object, and LocSpecs are assumed to automatically be part of an Anchor object.

It should also be noted that the OHP-Nav data model does not represent hypertext links as entities in their own right. I.e. there is no single OHP-Nav hypermedia object

within the OHP-Nav data model that corresponds to a single OHP-Nav hypertext link. This is because it is not necessary due to the dynamic retrieval method employed by the OHP-Nav protocol when traversing the hypermedia objects of a hypertext link (Section 3.2.6).

3.2.5. Linked List Representation

The OHP-Nav specification [Davis et al. 1996] and subsequent implementations (of OHP Navigation Servers) describe (or at least presume) OHP-Nav hypermedia objects to be connected to one another in a linked list formation [Reich et al. 1999a]. This is where hypermedia objects are embedded with (i.e. store) references of the other hypermedia objects to which they are attached. These object references are called association lists as they describe which hypermedia objects that a given hypermedia object is associated with:

- *OHP Link objects* are embedded with the references to *Endpoint objects*.
- *Endpoint objects* are embedded with references to *Anchor objects*.
- *Anchor objects* are embedded with references to *Node objects*.
- And *Node objects*, being the last item in the linked list, are not embedded with references to any other objects.

Figure 3.5 shows the object organisation of a typical OHP-Nav hypertext link commonly used within hypermedia literature.

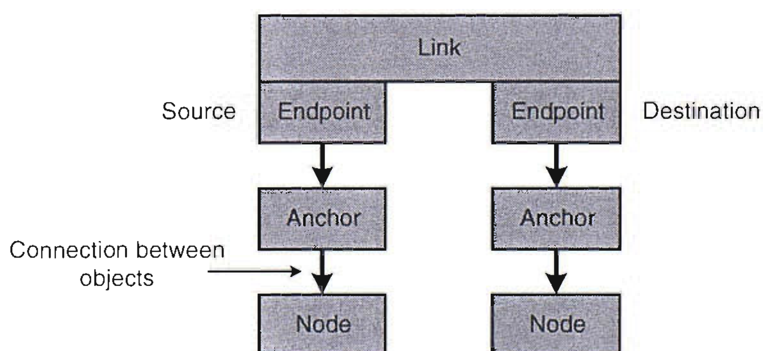


Figure 3.5: Common representation of an OHP-Nav hypertext link.

However, in order to depict the differences between the conventional object reference embedding approach and the OCS data model, this thesis has amended the

diagrammatic representation of OHP-Nav objects. The new representation of the OHP-Nav hypertext link of Figure 3.5 is shown in Figure 3.6.

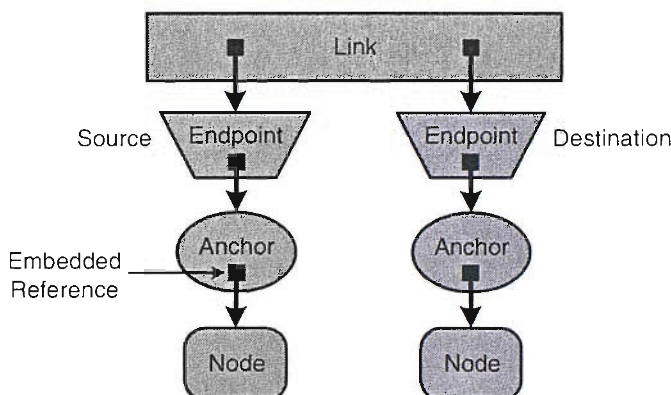


Figure 3.6: Amended representation of OHP-Nav hypermedia objects.

There are three key differences with the new diagrammatic representation:

1. *Hypermedia object shapes.* Each OHP-Nav object type is assigned a unique shape: OHP-Nav Link objects are elongated rectangles; Endpoint objects are isosceles trapezoids; Anchor objects are ovals; and Node objects are small round-edged rectangles. The reason for doing this is to ease identification of different OHP-Nav hypermedia objects when needing to refer to individual objects within thesis diagrams.

2. *Endpoint positioning.* Endpoint objects are physically separated from their associated Link objects. The reason for this change is because for an Endpoint to be physically touching a Link object implies that the lower object (the Endpoint) is wholly contained (i.e. embedded) within the Link object. (The idea of object embedding is further explained in Section 3.3.4). But OHP-Nav Endpoint objects are first class objects (Section 3.2.3), meaning that they are not embedded within any other hypermedia objects. Hence Endpoint objects are now depicted as being physically separated from the objects with which they are associated (as is the case with all other OHP-Nav objects).

3. *Embedded associations.* The amended representation of OHP-Nav hypermedia objects highlights the linked-list formation between OHP-Nav objects. This has been achieved by appending a box to one end of the arrows that represent the connections (i.e. associations) between objects. The box at the end of the arrow

indicates which object is embedded with the reference, and the arrow direction shows which object it is referencing.

Emphasising how association lists are embedded within hypermedia objects is extremely relevant, since the OCS data model identifies such embedding as a substantial weakness within the OHP-Nav hypermedia data model (Sections 7.2 and 8.2). As a result of this, the OCS data model adopts a different hypermedia object referencing scheme which is the focus of Chapter 7.

3.2.6. OHP-Nav Traversal

This section describes how OHP-Nav objects are used in their role of enabling navigation between documents. OHP-Nav objects are relevant since they are an example of the type of hypermedia objects organised by the OCS data model. Figure 3.7 shows the typical OHS components within a generic OHS environment. This acts as a useful reference when describing OHP-Nav object retrieval and traversal.

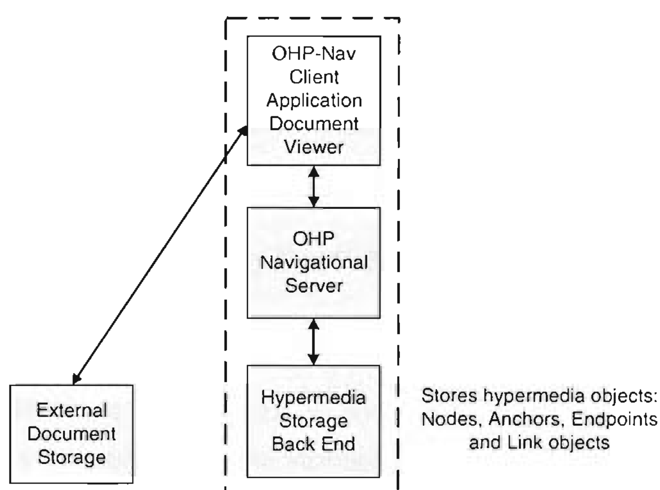


Figure 3.7: A generic OHP-Nav OHS environment.

3.2.6.1. Carrying Out Link Traversal

When a document is loaded into an OHP-Nav Client Application document viewer, one of the first activities undertaken by the viewer is to identify if any hypertext links reference the document as their source location. This is achieved by the viewer querying the OHP Navigation Server for Node objects. Section 3.2.6.2 describes how individual OHP-Nav objects (Nodes, Anchors, Endpoints and Links) are retrieved from the Navigation Server.

Once the Node objects have been retrieved, the next step is to find the Anchor objects that reference the retrieved Node objects. This enables the clickable anchor hotspots to be highlighted within the document. The viewer will then request the Endpoint objects that reference the retrieved Anchor objects in order to check that the hypertext links reference this document as their source location. At this point it is safe for the viewer to insert the highlighted anchor hotspots into the document content.

When the user clicks on one of the anchor hotspots in the document, the viewer will proceed to dynamically calculate which other hypermedia objects make up the associated hypertext link. The viewer will already know which hypermedia Anchor object corresponds to the clicked anchor hotspot and which source Endpoints are associated with the Anchor object as both sets of information will have been retained from when the hypertext link was first inserted into the document. Therefore the viewer will begin by finding the rest of the hypermedia objects that make up the hypertext link by querying the OHP Navigation Service for OHP Link objects that reference the source Endpoint objects. It will then find the destination Endpoint objects referenced by the Link objects. The viewer will then request the Anchor objects referenced by the Endpoint objects. And finally the viewer will query for Node objects that are referenced by the Anchor objects. Now the viewer can load the document referenced by the Node into the OHP-aware viewer.

3.2.6.2. Retrieving Individual OHP-Nav Objects

For each hypermedia object of an OHP-Nav hypertext link the Client Application must request an OHP-Nav object from the Navigation Server. The Navigation Server then requests the same object from the Storage Back End. The Storage Back End finds the object and returns it to the Navigation Server. The Navigation Server will then send the object to the Client Application.

Object reference embedding has a significant bearing on the ease with which individual OHP-Nav objects can be identified for retrieval. Of importance is which object type happens to record the connections between objects. For example the OHP-Nav linked-list representation (Section 3.2.5) stipulates that Anchor objects are responsible for recording the connection data that lists which Nodes are connected to which Anchors. Thus it is relatively easy to traverse from Anchor to Node since Anchor objects store the Anchor-to-Node connection data. Conversely it also means that it is more difficult to traverse from Node to Anchor since Node objects do not store the ID of the connected Anchor. Therefore given a Node object, the

Navigational Server has the extra responsibility of having to query the connection data content of all OHP Anchor objects (stored in the Storage Back End) in order to discover if any of those Anchors are connected to the Node in question. Such querying can be potentially time-consuming and processor intensive.

3.2.7. Implementations

Two prototype systems have been implemented to demonstrate the OHP-Nav protocol. These are Construct [Wiil 2001] and the Solent system. (Construct's architecture was previously discussed in detail in Section 2.15.1.)

It is the Solent system [Reich et al. 1999b; Reich et al. 1999a] that is of particular interest since the work I carried out on designing and programming the Storage Back End and Retrieval Server influenced the design of the OCS data model (Chapter 7 onwards). This is in respect of investigating the wasteful storage of duplicate hypermedia objects. Moreover a review of the Solent system provides an understanding of the environment within which the hypermedia objects organised by the OCS data model operate.

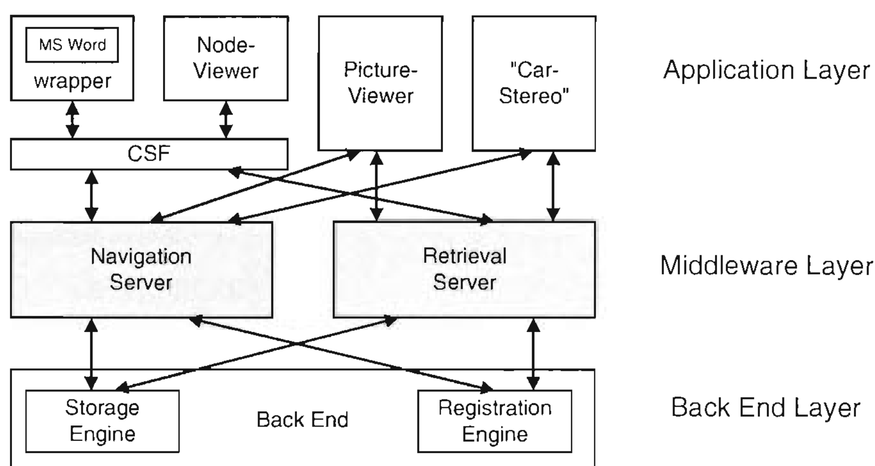


Figure 3.8: Conceptual architecture of the Solent CB-OHS.

Solent is a Component-Based OHS (Section 2.14) made up of several inter-communicating components (Figure 3.8). All Solent applications are enabled with OHP-Nav functionality. The Middleware Layer comprises a Retrieval Server and a Navigation Server. This facilitates communication between the Application and Back End Layers. The Back End comprises a Registration Engine and Storage Engine. The Registration Engine is a basic broker object with which other components can

register. The Storage Engine provides storage and retrieval of arbitrary structures encoded in XML (eXtensible Markup Language) which are stored in an SQL database.

3.2.7.1. Example OHP-Nav message

Figure 3.9 shows a typical OHP-Nav message. It is an example of a fragment of a message used to create an ANCHOR object encoded in XML.

```
<MESSAGE>
    <MESSAGEHEADER>
        <SENDER> Picture Viewer Application </SENDER>
        <RECEIVER> Navigation Server </RECEIVER>
        <SERIAL> S001 </SERIAL>
        <MNAME> ANCHORCREATE </MNAME>
        <PROTOCOL> OHP-NAV-1.0 </PROTOCOL>
        <CONTEXTIDSET>
           <CONTEXTID> C101 </CONTEXTID>
        </CONTEXTIDSET>
        <PERFORMATIVE> ask </PERFORMATIVE>
    </MESSAGEHEADER>
    <MESSAGEBODY>
        <ANCHORCREATE>
           <ANCHOR>
             <ID> A1 </ID>
             <PARENTID> N1 </PARENTID>
             <AXISLOC>
               <FWDAXISSET>
                 <AXIS>
                   <NAME> characters in </NAME>
                   <TYPE> CHAR </TYPE>
                   <VALUESET>
                     <VALUE> 109 </VALUE>
                     <VALUE> 118 </VALUE>
                   </VALUESET>
```

```
                    </AXIS>
                  </FWDAXISSET>
                  ...
                </AXISLOC>
              </ANCHOR>
          </ANCHORCREATE>
      </MESSAGEBODY>
</MESSAGE>
```

Figure 3.9: OHP-Nav message to create an Anchor object.

Every OHP-Nav message is split into two parts, a MESSAGHEADER and a MESSAGEBODY.

The MESSAGEHEADER tags are the same per message:

- The SENDER tag specifies which component is sending the message.
- The RECEIVER tag specifies which component is receiving the message.
- The SERIAL tag is the unique ID of the message.
- The PROTOCOL tag specifies the protocol that the MESSAGEBODY follows. In this case it is an OHP-Nav protocol formatted message, other examples may be an OHP workflow or computation message.
- The PERFORMATIVE tag describes how communication takes place between components. A few examples are listed:
  - *ask*. Used to ask a component to perform a service, e.g. to create an Anchor object (as is the case with the example XML message fragment shown).
  - *notify*. Issued by the recipient component of an *ask* request. The MESSAGEBODY reports the success (or lack of) on performing an *ask* request.
  - *register*. This is sent by a component to let all other components know that this component exists and is available for sending and receiving messages.
  - *advertise*. Issued by one component to inform all other components that they can subscribe to certain types of messages that emanate from the issuing

component. For example a component may *advertise* that it can let other components know when it has successfully created a new Anchor object.

- *subscribe*. Sent in response by other components to an advertise request to let the advertising component know that they are interested in being informed of the advertisers' advertised event.

The actual contents of the OHP-Nav message are contained in the MESSAGEBODY. It describes how an OHP-Nav object (Node, Anchor, Endpoint, Link, Context or PSpec) is to be created, deleted, modified or retrieved (e.g. LINKCREATE, ENDPOINTDELETE, ANCHORMODIFY, NODERETRIEVE, etc.). Or it can be used to declare whether an OHP-Nav object has been successfully created, deleted, modified or retrieved (e.g. LINKCREATED, ENDPOINTDELETED, ANCHORMODIFIED, NODERETRIEVED, etc.).

The MESSAGEBODY of the example XML fragment of Figure 3.9 defines how to create a new Anchor object, hence the MESSAGEBODY contains Anchor-specific data:

- The anchor is assigned unique identifier A1.
- The PARENTID tag is an example of an embedded object reference as it specifies, within the anchor object itself, to which OHP-Nav object this anchor is associated with. In this case it is Node object N1.
- The AXISLOC tag specifies the location of the hotspot of the link within the connected node, i.e. it is to span characters 109 to 118 within the content of Node N1.

3.2.8. OHP Demonstrations

Both the Solent and Construct CB-OHSs were demonstrated at the ACM Hypertext '98 and Hypertext '99 Conferences.

3.2.8.1. Hypertext '98 Demonstration

The rationale behind the Hypertext '98 demonstrations was to show interoperability between clients of different Open Hypermedia Systems for the very first time [Millard et al. 1998a; Bouvin 2000; Millard et al. 2000a]. The demonstrations brought

together several clients of the Solent and Construct OHSs that implemented the OHP-Nav protocol. Four interoperability capabilities were demonstrated:

1. *Applications devised for one OHS being able to communicate with other OHS servers.* This was demonstrated by the Solent Picture-Viewer (client of the Solent CB-OHS) working with the Construct Hypermedia Server.

2. *Applications of different OHSs retrieving the same document.* This was demonstrated by the Solent Picture-Viewer and the Construct Emacs client being connected to the Construct server and simultaneously retrieving the same document.

3. *The ability to author links between client applications that belong to different OHSs.* This was demonstrated by one Endpoint object being created using the Solent Picture-Viewer, and a second Endpoint object being created using the Construct Emacs client. The Solent Link-Editor was then used to author a link between the two Endpoints.

4. *The ability to follow a link created by client applications that belong to different OHSs.* This was demonstrated by a user of the Construct-adapted Microsoft Internet Explorer clicking and following the link created in Demonstration 3.

The main emphasis of the demonstrations was to show that through a standardised interface any OHS client application can control and access the structure and content maintained by another OHS (Demonstrations 1, 2 and 4). The demonstrations also showed that it was possible to connect to hypermedia structure created by another OHS client and managed by another OHS (Demonstration 3).

3.2.8.2. Hypertext '99 Demonstration

The demonstrations of the Hypertext '99 Conference built on top of the Hypertext '98 demonstrations and were used as an opportunity to show the flexibility of both the CoReArc framework (Section 3.2.1) and the Core Data Model (Section 3.2.2). This was achieved by expanding the Solent CB-OHS to use the Computational Interface (Section 3.2.2.2).

The Hypertext '99 demonstrations centred on showing how the CoReArc framework could readily contain new client applications at the client application layer (e.g. the "Car Stereo" client) and contain new servers at the middleware layer (e.g. the Retrieval Server). It was also demonstrated how new object classes could be added to the Core Data Model (e.g. the Retrieval Class) which in turn enables the creation of

new protocols. For example the introduction of the Retrieval Class led to the Computation protocol being developed.

Both sets of demonstrations at both conferences were successful, and were met with great acclaim by the hypermedia community at large.

More importantly for my research was that through developing the server and storage components for the Solent CB-OHS, it made me appreciate the amount of wasteful and inefficient use and storage of general hypermedia structure that the OHP-Nav data model unwittingly encouraged (Sections 6.2 and 6.3). It was this that led me towards developing the OCS data model.

3.3. The Fundamental Open Hypermedia Model

The Fundamental Open Hypermedia Model (FOHM) [Millard 2000a] is a data model developed at Southampton University that was designed to investigate the common features of interoperability within the navigational, spatial and taxonomic hypertext domains. It therefore presents a further platform upon which to explore the OCS data model as it provides access to a generic model of hypermedia structure enabling the OCS data model to be applied to the spatial and taxonomic hypermedia domains as well as the traditional navigational hypermedia domain.

However my initial interest in FOHM, as regards the OCS data model, sprung from experimentation with FOHM structure versioning [Griffiths et al. 2002] which is explained in Section 6.4.2.

3.3.1. FOHM Background

FOHM came about as a result of OHP work on Component-Based OHSs. In the CB-OHS architecture (Figure 3.1), OHSs implement middleware components for each hypertext domain. These components are responsible for mapping the hypermedia structures, required by clients, into and out of the structures stored in an all purpose back-end server. This means that each middleware component must be adapted so that it can translate the structures in all other domains into its own domain. The problem is that there are an open number of domains, theoretically in the hundreds and upwards, that a middleware component must be adapted to interoperate with.

FOHM was devised so that such adaptations of middleware components are unnecessary. FOHM research investigates the capability of one hypertext domain being able to interpret a second hypertext domain as if it was the first. For example using a navigational browser to follow a hypertext link from a navigational workspace into a spatial workspace whereby the navigational browser would attempt to interpret the spatial workspace as if it was a navigational workspace [Millard et al. 2000b]. To this end the intent behind FOHM is to define a common storage layer sufficiently generic so that it can store the structures required by all current and future domains.

3.3.2. Hypermedia Domains

FOHM's remit is to investigate interoperability between the navigational, spatial and taxonomic hypermedia domains.

3.3.2.1. Navigational Hypermedia Domain

The navigational domain is the most well known hypermedia domain. It is the domain of the World Wide Web [Berners-Lee et al. 1992], and has been the main focus of the majority of the OHSWG's efforts taking the form of OHP-Nav. Figure 3.6 shows an example of a typical OHP-Nav hypertext link.

3.3.2.2. Spatial Hypermedia Domain

Spatial hypermedia allows users to organise their information visually such that relationships between node object resources[4] are expressed by their visual characteristics, e.g. proximity, colour or shape. Spatial hypermedia expresses classification within nodes where some nodes can be said to be more related than others. For example a node slightly misaligned with other nodes might express uncertainty as to whether the node is actually part of the relationship. Examples of spatial hypermedia systems include VIKI [Marshall and Shipman 1997] and CAOS [Reinert et al. 1999].

---

[4] A node in this instance means a referenced arbitrary resource rather than an OHP Node which is a proxy object for a given resource.

3.3.2.3. Taxonomic Hypermedia Domain

Taxonomic hypermedia allows the same pieces of information (called artefacts) to be categorised into different views (called perspectives). Users can navigate the taxonomic hierarchy by moving between overlapping sets and can reason about the relationships that node object resources have with one another.

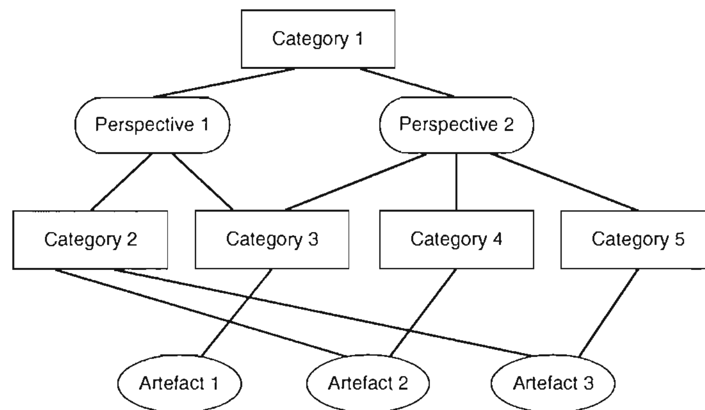


Figure 3.10: Example Taxonomy.

Figure 3.10 provides a taxonomic hierarchy example (adapted from [Millard et al. 2000b]). Two users have categorised three artefacts. But they have different opinions on how the three artefacts ought to be split. The two perspective objects represent the different categorisations for the three artefacts.

3.3.3. FOHM and Structural Computing

FOHM's role within the Structural Computing environment is as a Content Handler Interface supported by a dedicated structure server. This is reflected by FOHM's implementation as a structure server, called Auld Linky (Figure 3.11) [Michaelides et al. 2001; Griffiths et al. 2002; Millard 2003]. It is a simple stand-alone server that stores hypermedia structure represented in the FOHM format that serves FOHM structures expressed in XML. Queries are sent to Linky in the form of a FOHM XML pattern that is matched against Linky's stored structures. Those that match are then returned.

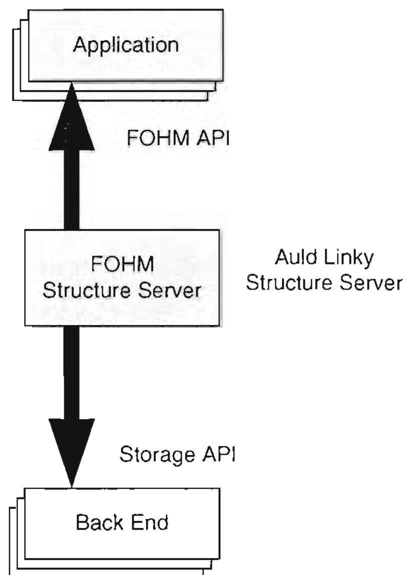


Figure 3.11: Auld Linky placement within a Structural Computing environment.

FOHM's ability to model generic structure means that it can utilise the structural characteristics of one hypermedia domain to transform the representation of hypermedia structure within another domain [Millard and Davis 2000]. For example anchors (of the navigational hypermedia domain) can improve the granularity of referencing (i.e. pointing) within spatial or taxonomic hypermedia structures.

Spatial domain structures (such as sets, queues, lists and stacks) can also be used to enhance traversal in the navigational hypermedia domain. For example the endpoints of a link in the navigational domain are ordinarily arranged as a set. During typical link traversal users arrive at all endpoints except the starting one. FOHM can change this experience by allowing navigational endpoints to be structured differently, e.g. if arranged as a stack, users would arrive at those endpoints either side of the starting endpoint.

3.3.4. The FOHM Data Model

The FOHM data model is essentially composed of four object types: Associations, Bindings, References and Data objects. They are very similar to the four basic object types found in the OHP-Nav data model (Section 3.2.4).

- *Association* (first-class object). Represents a relationship between Data objects or other Associations. An Association can contain any number of Bindings, but it may only be connected to at most one Reference object. Every Association contains a feature space which is a list of features that all the objects of the

Association must map to. Like OHP-Nav Link objects, FOHM Associations are depicted as elongated rectangles.

- *Binding* (embedded within Association objects). Attaches References to Association objects. A Binding can be contained by at most one Association and be connected to at most one Reference object. Bindings also contain a feature value of the Association object's feature space. Bindings are diagrammatically depicted as semi-circles.

- *Reference* (first-class object). Points at or into Data objects or Association objects. A Reference object can be connected to at most one Binding and either connected to one Data or Association object. FOHM References are depicted as circles.

- *Data* (first-class object). An object that serves as a wrapper for some piece of data that lies outside the scope of the model. For example a document, file or streamed data. A Data object can be connected to one Reference object only. FOHM Data objects are depicted as small rectangles.

Figure 3.12 shows an example of a FOHM structure: a navigational link that points to a spatial list. The top-most Association is a navigational link since it has a direction feature space and its Bindings contain navigational directions. The bottom-most Association represents a spatial relationship whose Data objects are ordered in a list (indicated by each Binding containing a numerical value).

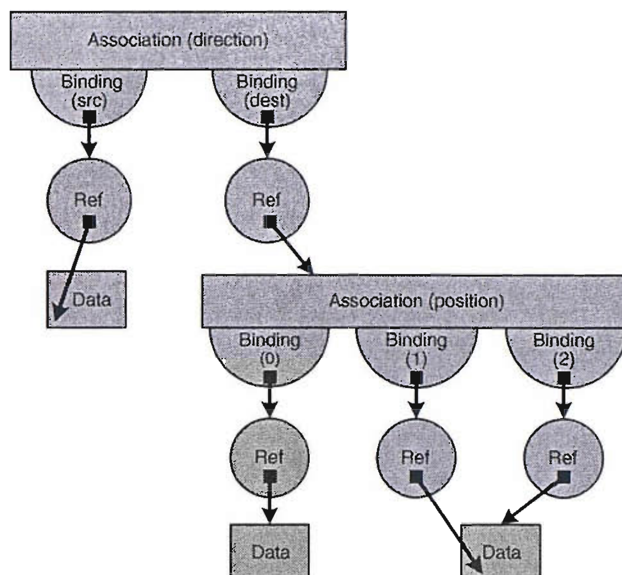


Figure 3.12: Example FOHM structure.

Connections between FOHM's objects bear similarities to OHP-Nav, but at the same time FOHM's connections are subtly different. Only three of FOHM's four object types are primary objects. These are Associations, References and Data objects. Bindings are not primary objects as the FOHM data model explicitly prescribes that they are always wholly contained within Associations.

Figure 3.12 shows Binding objects to be wholly contained (i.e. embedded) within Association objects by the absence of a connection arrow between Association and Binding objects, and that the Binding object physically touches the bottom of the Association object.

As was the case with the OHP-Nav representation of hypermedia objects, so too the thesis has amended the connections between FOHM objects. These changes have already been shown in Figure 3.12. The amendment takes the form of appending a box to one end of the connection arrow to indicate which object contains the embedded referencing object information.

Connections between FOHM objects generally take one of two arrangements:

1. One FOHM object can contain an embedded (i.e. stored) reference to the other FOHM object (as is the case with the linked list arrangement for OHP-Nav objects). The uni-directional arrows between FOHM objects in Figure 3.12 indicate which objects store the embedded reference enabling association with a secondary object.

2. Alternatively any FOHM object can be wholly contained within another FOHM object. For example a Reference being wholly contained by a Binding object.

This act of embedding references and/or wholly containing objects has a negative effect on hypermedia structure re-use and hyperstructure versioning. Coupled with the OHP-Nav embedding of object references, it was with these problems in mind that the OCS data model was developed (Chapter 7 onwards).

3.4. Summary

This chapter has described the OHP-Nav and FOHM data models. OHP is an OHS interoperability project that has led to standardized interfaces (e.g. OHP-Nav), a conceptual OHS architecture (the CoReArc model) and standardised data models (e.g. the OHP-Nav data model). OHP-Nav provides a good vehicle for experimenting with hypermedia structures of the navigational domain, and by working with members of the OHSWG to devise the Solent CB-OHS it brought to my attention the problematic linked-list representation of connections between OHP objects, hence the need for the OCS data model.

Continuing in the same vein as the OHP is the FOHM research project. It investigates navigational, spatial and taxonomic inter-domain interoperability. It aims to benefit the construction of hypermedia middleware servers so that each server is not required to translate the structures of different domains into its own domain. It was through experimentation with versioning FOHM structure (Section 6.4.2) that (again) highlighted the deficiencies in the linked-list representation of connections between structural objects.

Chapter 4. Versioning Background

4.1. Introduction

This chapter sets the background to hypermedia versioning. Selected are a range of systems that adopt different strategies for versioning hypermedia structure (Section 4.5). Also examined is the problem of revision proliferation (Section 4.6). These are under investigation since the OCS data model not only provides an improved framework within which to version hypermedia structure (Chapters 9 and 10), but it also addresses the problem of revision proliferation (Chapter 9) too.

4.2. What is Versioning?

Versioning is the act of organising, co-ordinating, managing and tracking the development of the same changing resource at different stages of its evolution. The hypermedia versioning systems of this chapter version either just node data (e.g. documents), hypermedia structure (whole hypertext links and/or the individual hypermedia objects that comprise links) or a combination of both (e.g. hypertext networks composed of both nodes and links).

As regards the OCS data model, its interest lies in versioning individual hypermedia objects, the connections between hypermedia objects (of a hypermedia structure) and hypertext structures as a whole (e.g. links). It does not include the versioning of node resources referenced by versioned structures.

4.3. Versioning Policy

The typical versioning policy whenever an attempt is made to update an existing hypermedia resource is to preserve (or freeze) it. This is by creating a new copy (i.e. a new revision) of the resource. It is upon this copy that the update can be enacted. Meanwhile the original copy of the hypermedia resource remains untouched, hence it is preserved.

4.4. Why Version Hypermedia Resources?

Many benefits are afforded by versioning hypermedia resources:

- Enables hypermedia resources to be preserved as they evolve over time.
- Access to previous revisions of hypermedia resources.
- Hypermedia resources (e.g. hypertext networks, nodes or links) can be rolled back to their previous versioned state after changes have been made to them. This offers a failsafe baseline for experimentation [Østerbye 1992].
- The maintenance of alternative revisions. This is again useful for experimentation.
- Enables evaluation of developmental progress through comparing different revisions [Bendix et al. 2001].
- Provides a historical record of a hypermedia resource's evolution.
- Supports hypermedia collaborative work when needing to freeze revisions where many users are working on a shared resource [Whitehead 2001b].
- Useful in application areas where the connections represented by hypermedia structures cannot afford to be broken such as inter-document relationships within legal and audit documents [Whitehead 2001b].
- Offers solutions for the problem of link maintenance through position tracking (see Chapter 5).

4.5. Hypermedia Versioning Systems

This section provides a summary of well known hypermedia versioning systems. It is not an exhaustive list since there have been many hypermedia versioning systems over the past 20 years [Vitali 1999]. Particular systems have been selected because they represent different approaches to versioning and/or they suffer some form of revision proliferation (this term and how it affects particular versioning systems is discussed in Section 4.6).

The section begins with an overview of Jim Whitehead's Hypertext 2001 paper [Whitehead 2001b] which describes the different approaches for the storage of hypermedia revision histories. These are an essential part of a hypermedia version system because without some form of recorded version history, much versioning functionality is lost. Whilst revision objects would still be preserved, there would be no way of knowing how or even if one revision is derived from another.

The subsequent subsections make reference to this terminology to describe the hypermedia versioning systems under review.

4.5.1. Design Spaces for Versioning

Whitehead identifies three strategies for recording the version histories of hypermedia resources. These are:

- *Versioned-Object approach.* This approach records each revision of a resource as a separate object. The version history is recorded by a separate container, and the relationship between version history container and members is referential, often recorded by the container itself. The predecessor/successor relationships between revisions are either embedded within the members or recorded within separate first-class relationship entities. If the latter case is true, then these objects are also referentially contained by the version history container. The advantage of the Versioned-Object approach is that revisions can belong to other containers beside the original version history container, e.g. configurations.

- *Within-Object approach.* This approach utilises a version history container that physically includes all derived object revision members. This has the advantage that all revisions are locked within a single object, thereby guaranteeing the stability of references to these objects. But the disadvantage is that revisions cannot participate in other containment structures, unless a replica of a specific revision is created and placed into the other container. An example of such a system is RCS [Tichy 1985].

- *Independent Relationship Object approach.* This approach records the version history predecessor/successor relationships as separate first class objects. Hence there is no container set that represents a version history as a single entity. The advantage is that there is less overhead since no version history containers need updating whenever a new object revision is derived from an existing revision. The disadvantage is that referential integrity cannot be guaranteed when attempting to retrieve all revision objects of a version history. For example some relationship

entities may be located across an organisational boundary that is not accessible when building the version history. Hence the entire version history may fail to be built.

Regarding the recording of the revision histories of hypertext links, if they are independent entities (i.e. separate from their referencing nodes), then their revision history can be recorded by any of the three above strategies. But for links wholly embedded (i.e. contained) in node resources, when the node is versioned so too is the link. Therefore the version history of embedded links will be the same as that of the nodes that contain them.

4.5.2. Xanadu

One of the earliest systems to consider hypermedia versioning was Project Xanadu (Section 2.7.3). It operates an implicit form of versioning, i.e. it implicitly enables versioning to be carried out without the system having to expressly make versioning commands available for clients.

Xanadu versioning works by new revisions of Xanadu documents directly re-using the content that is the same between old and new revisions. This is through the latter document revision's content list pointing at the same content list referenced by the old revision. Any new content for the new revision is assigned its own permanent address, and this is included within the new revision by its content list also pointing at the permanent address of the new content.

Xanadu's version history is represented by the Independent Relationship Object approach. Transclusion links (not embedded in content lists) [Nelson 1999a] indicate how content lists, and the content referenced by content lists, are derived from one another.

4.5.3. HyperPro

HyperPro [Østerbye 1992] is a hypermedia system that versions nodes, but not links (at least not explicitly). It is an example of the Versioned-Object approach since individual versioned nodes are referentially captured within version sets, a form of composite. HyperPro can also capture the state of an entire hypertext network within contexts, another form of composite. Contexts referentially group together version sets whose revisions (within the version sets) make up a hypertext network. A hypertext network is versioned by versioning contexts. In this way HyperPro can

support implicit limited link versioning since any links located inside versioned contexts are versioned too.

Figure 4.1 shows an example adapted from [Østerbye 1992]. Context G1 is the initial state of the hypertext network where A, B, C and D are version sets containing individual node revisions interconnected by version links. Contexts G2 and G3 show later revisions of the hypertext network. New node revisions (within version sets) have been added as well as new interconnecting links.

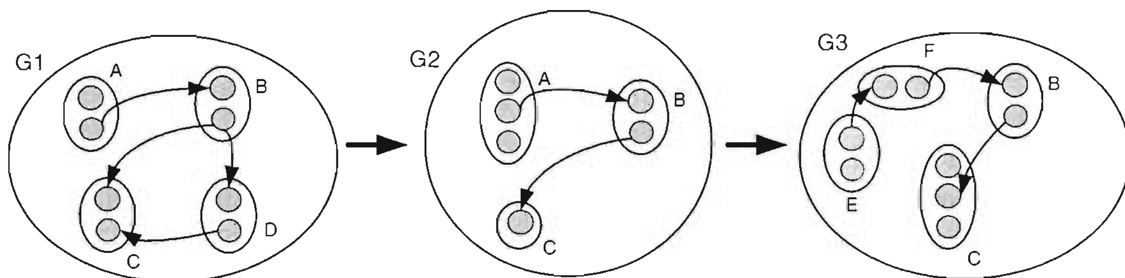


Figure 4.1: Versioning HyperPro contexts.

Node revisions can be selected one of two ways: Generic Version Links point to entire version sets. They compute their destination node by retrieving the node identified by the context's recorded selection criteria, e.g. latest revision. Alternatively a Specific Version Link can be used to point to specific node revisions inside version sets.

4.5.4. CoVer

CoVer is a separate hypermedia version server designed to assist the authoring of hypermedia documents [Haake 1992; Haake and Haake 1993; Haake 1994]. Its difference is that it combines task- and state-based versioning within a single conceptual model.

Task-based versioning involves breaking a job (e.g. writing a paper) into multiple tasks (e.g. writing each individual chapter sequentially or in parallel). A task is a composite that holds references to subtasks and/or objects that together determine the task's current state. Objects within a task can be a combination of nodes, e.g. documents or document segments, links or composites. The role of tasks is to monitor the revisions of the various objects used and created in the context of performing a job.

CoVer uses state-based versioning to record every legal state of the versioned nodes, links and composites that are used to complete a task. It also groups each set of derived objects within version sets which also record their evolution history. CoVer's use of referential containers, e.g. tasks and version sets, mark it out as adopting the Versioned-Object approach to versioning.

CoVer can freeze nodes in their entire state. But freezing the state of a link or composite means implicitly freezing any other objects (e.g. nodes, links or composites) referenced by the frozen link/composite. This can lead to revision proliferation as described in Section 4.6.2.

4.5.5. DeltaV

The DeltaV protocol [Whitehead 2001a; Kim et al. 2004] builds on the features and data model of the WebDAV (Web-based Distributed Authoring and Versioning) protocol [Whitehead and Goland 2004]. It provides an open, standards-based infrastructure to support the versioning of Web pages through extending the HTTP protocol.

DeltaV is an example of a versioning system for the Web. It can record the individual states of evolution of a Web page, and it can record a snapshot of the current revisions of multiple Web pages. The latter is equivalent to being able to freeze the state of a hypertext network. The DeltaV protocol cannot version hypertext links as separate entities since they are embedded within Web pages. But embedded hypertext links can be implicitly versioned when the containing Web page is versioned. Hence the history of a versioned Web page's hypertext links is the same as the history of the versioned Web page itself.

4.5.6. HyperProp

HyperProp [Soares et al. 1993a] is a hypermedia system designed to provide an environment for hypermedia application construction. Its data model, which revolves around the Nested Composite Model (NCM), marks HyperProp and its approach to versioning as different because the NCM implements a specialist composite called the User Context Node. It relates individual nodes (e.g. text, graphic, audio along with other User Context Nodes) to each other by containing one node within another through embedded object reference linking.

For example, to organise a textbook, a User Context Node may be created which represents the overall textbook. It may also contain a set of User Context Nodes that represent chapters, and each chapter may contain a further set of nodes which represent chapter sections. Hence in this way nodes are said to be nested within one another. Figure 4.2 shows an example.

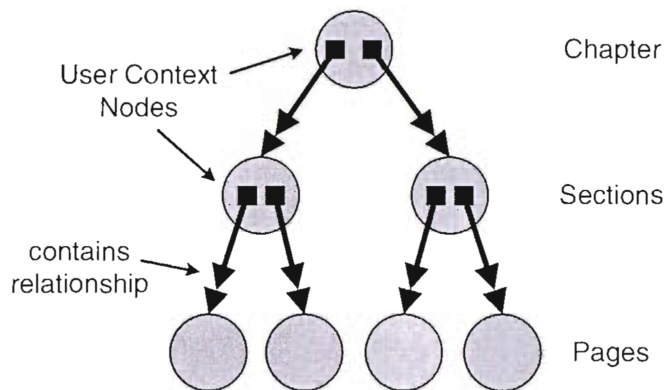


Figure 4.2: User Context Nodes of the HyperProp Nested Composite Model.

The NCM [Soares and Casanova 1994; Soares et al. 1999] enables individual nodes, including User Context Nodes, to be versioned. But it does not version hypertext links. However the objects and relationships of a hypertext network can be represented by a User Context Node which can be frozen. Hence any links connecting nodes within the network will also be implicitly frozen. But freezing a User Context Node means capturing the current state of all objects contained by that User Context Node. The drawback with this approach is that it causes revision proliferation as explained in Section 4.6.3.

4.5.7. Hypermedia Versioning Control Framework

The Hypermedia Versioning Control Framework (HVCF) [Hicks et al. 1998] is one of the few systems that offer complete version control for both documents and open hypertext links (i.e. hypertext links that are stored separate from document content). The HVCF takes the form of a hypermedia versioning server incorporated as part of the HB3 Open Hyperbase Management System [Leggett and Schnase 1994]. It provides versioning as an "all-or-nothing" approach [Østerbye 1992], i.e. every object (data or structure) of HB3 is versioned, and all object manipulation requests automatically pass through the version server.

The HVCF enables applications to select basic operations offered by the version server, e.g. creation, deletion, update and retrieval operations for individual revisions. These are then used to implement application-specific versioning policies that give applications full control as to when and which objects (or groups of objects) are versioned. This can apply to individual hypertext links, or an entire hypertext network. The latter is versioned by collecting links and documents into composites and then versioning them.

The HVCF is an example of the Versioned-Object approach since every HB3 object is automatically assigned to a version set composite which records object evolution. The HVCF also employs a delta storage algorithm technique in order to reduce storage requirements for all object revisions.

4.5.8. Chimera Versioning

The Chimera approach to versioning [Whitehead et al. 1994] is an example of a scheme that versions hypermedia structure only, and not the data (i.e. documents) they relate. Thus ensues a not unrealistic scenario where data objects and hypermedia structure are stored and versioned independently of one another. This is not dissimilar to the OCS data model which since it models hypermedia structure only, is also concerned with versioning hypermedia structure only.

Chimera versioning focuses on the use of configuration objects. These are a restricted form of composite object that record collections of pointers to hypermedia structure revision objects. They name, collect and version any subset of hypermedia structure that might be affected by modification to an external data object. For example recording the anchors that reference an external document's content. This makes it easier to identify the new structural revisions that must be created to make sure that any external data objects continue to reference their intended resources via Chimera structures. Chimera is an example of the Versioned-Object approach to versioning structure.

4.6. Revision Proliferation

Revision proliferation is the generation of unintended object revisions when an existing revision is being modified [Conradi and Westfechtel 1998]. It is a recurring problem within hypermedia resource versioning.

Three example cases of revision proliferation are highlighted in HyperPro, CoVer and the Nested Composite Model. Two further examples of revision proliferation occurring in OHP-Nav and FOHM are also described in Section 6.4. Revision proliferation was one of the four problem domain issues that led to development of the OCS data model (Sections 6.4.1 and 6.4.2.4).

4.6.1. HyperPro

Østerbye in the same paper that discusses HyperPro [Østerbye 1992] also discusses circumstances that can lead to revision proliferation. This is where versioning is permitted on nodes only, and links are semantically considered part of the node (whether physically embedded or not) such that every link change constitutes a change to the node. Figure 4.3 shows an example. (Dashed lines represent version sets.)

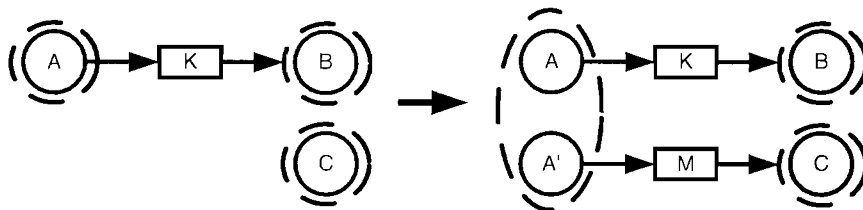
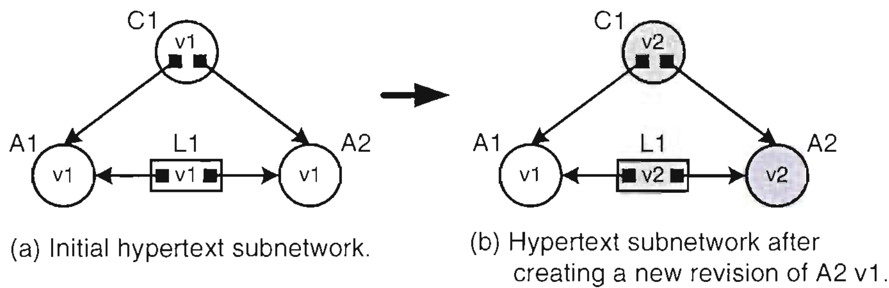


Figure 4.3: HyperPro revision proliferation scenario.

Hypertext link K has frozen node A as its source. If K is modified so that it is re-directed from node B to C, then a new copy of link K is created shown as link M. Also, a new copy of node A is created since changing a link associated with a frozen node amounts to an attempt at changing that node. This is an example of revision proliferation since new node revisions are created, but the only difference between the previous and new node revisions is that the associated links have been changed, not the actual node content itself.

4.6.2. CoVer

This example of revision proliferation is based on a CoVer example from [Haake 1994]. Figure 4.4(a) shows an initial hypertext subnetwork. Composite 'C1 v1' contains two nodes: 'A1 v1' and 'A2 v1'. They are connected to one another by independent link 'L1 v1'.



(a) Initial hypertext subnetwork.

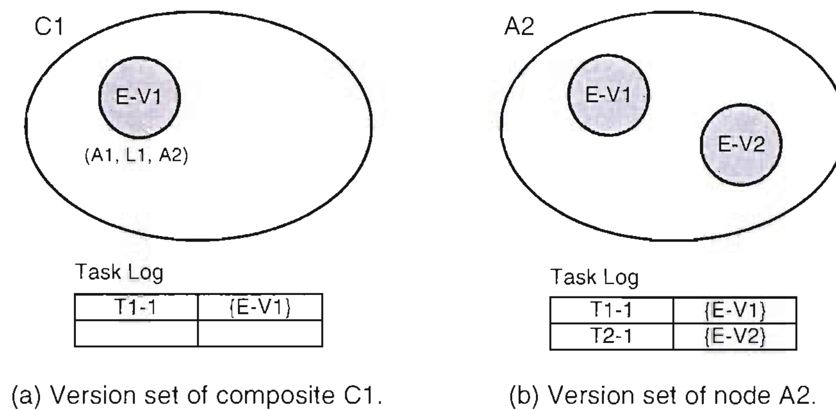
(b) Hypertext subnetwork after creating a new revision of A2 v1.

Figure 4.4: CoVer revision proliferation example.

Revision proliferation (i.e. the creation of unnecessary revision objects) occurs when the client elects to create a new revision of A2, shown as 'A2 v2' in Figure 4.4(b). In order to maintain the connections within the hypertext subnetwork new revisions of link L1 and composite C1 are automatically created. 'L1 v2' is necessary to connect new 'A2 v2' to existing 'A1 v1', and 'C1 v2' is needed to contain the new structural arrangement. However the automatic creation of the new objects should not really be necessary since the only difference between the old and new revisions is that the new revisions now reference 'A2 v2' instead of 'A2 v1'. No worthwhile object content update has taken place.

The designers of the CoVer system have devised their own system to tackle the problem of revision proliferation. This is by CoVer only recording explicit revisions and omitting implicit revisions. Explicit revisions are the new revisions deliberately intended by the client, e.g. 'A2 v2'. Implicit revisions are the new revisions automatically created as a side-effect of creating the explicit revisions, e.g. 'C1 v2' and 'L1 v2'.

The CoVer solution is shown in Figure 4.5. This time only the version sets of the revision objects of Figure 4.4 are shown. Figure 4.5(a) shows the state of the version set of composite C1 both before and after node A2 has been modified (previously indicated by Figures 4.4(a) and (b)). The ellipse depicts the version set of composite C1. The entity within the version set marked Explicit-V1 indicates that the version set only comprises v1 of C1 and is an explicit object. The explicit entity contains the version sets of A1, L1 and A2. Version Set C1 also possesses a task log. It records within which task the explicit object v1 of C1 was created.



(a) Version set of composite C1. (b) Version set of node A2.

Figure 4.5: CoVer revision proliferation solution.

Figure 4.5(b) shows the CoVer representation of A2 after it has been modified. A2's version set (the ellipse) comprises two revisions. The task log indicates that the first revision of A2 (Explicit-V1) was created as part of task T1-1, i.e. at the same time as 'C1 v1'. The second revision (Explicit-V2) was created as part of later task T2-1.

The implicit revisions (i.e. 'L1 v2' and 'C1 v2') generated by the creation of explicit revision 'A2 v2' can be calculated by combining the task log data, explicit objects and making certain versioning assumptions. The assumptions are that when the content of a composite is changed, a new revision of it will be created; and that when the endpoint of a link is changed, a new revision of that link will also result.

Therefore in order to swap 'A2 v2' in place of 'A2 v1', the following assumed versioning actions occur: 'L1 v1' is updated to enable it to reference a new revision of A2, and 'C1 v1' is updated so that it contains new revisions of A2 and L1.

However this solution does not completely eradicate the problem of revision proliferation. This is because the implicit objects (generated as a result of revision proliferation) are still necessary in order to describe the state of the hypertext subnetwork after 'A2 v2' has been created. Instead the CoVer solution only saves on the storage of implicit objects. But this is at the expense of having to implement a much more complicated scheme that just reproduces the implicit objects whenever a client wants to explore the state of composite C1 when it contains 'A2 v2'.

Furthermore the CoVer explicit object solution also makes pattern matching more complicated to perform. For example in order to match a hypermedia pattern against all structures stored by the CoVer hypermedia system, then CoVer must undergo the complicated process of re-building each and every hypermedia structure based on the explicit object and task log data in order to carry out such a comparison.

Section 10.6.1 presents the OCS data model solution which eradicates the revision proliferation problem as depicted within the CoVer paper.

4.6.3. HyperProp

The developers of the Nested Composite Model also recognised the dangers of revision proliferation [Soares et al. 1993b]. In this case the problem manifested itself when an attempt is made to modify a frozen node which is a member of one or more frozen nested composites. Usual versioning policy dictates that new revisions would be created of all the nested composite nodes, but a potentially large number of undesirable (and seemingly unnecessary) nodes would result.

Figure 4.6 illustrates the problem. The initial hypertext network is shown in Figure 4.6(a) where node 'D v1' is contained by User Context Nodes 'E v1', 'B v1' and 'F v1'; 'E v1' is also contained by 'C v1', and 'B v1' is contained by 'A v1'. The problem is that the creation of revision 'D v2' leads to the generation of five new User Context Nodes as shown by Figure 4.6(b).

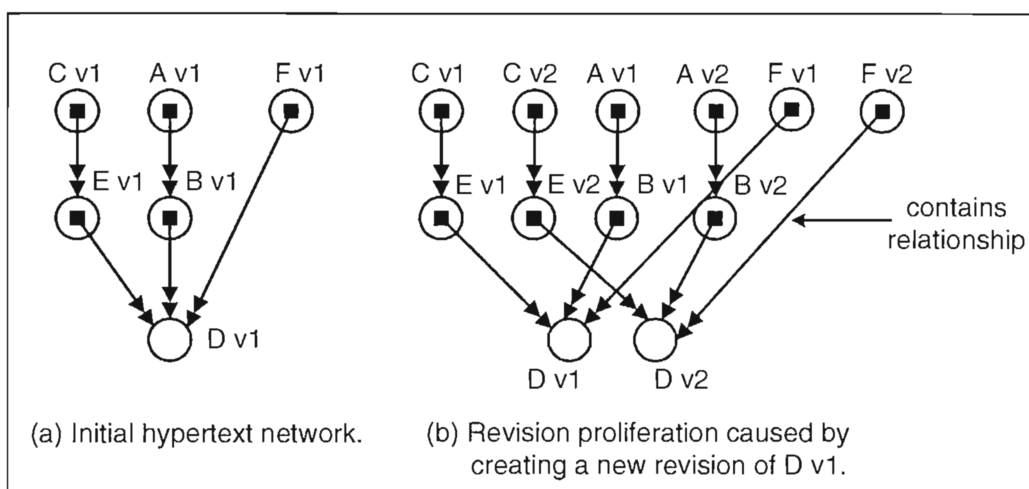


Figure 4.6: Revision proliferation in NCM.

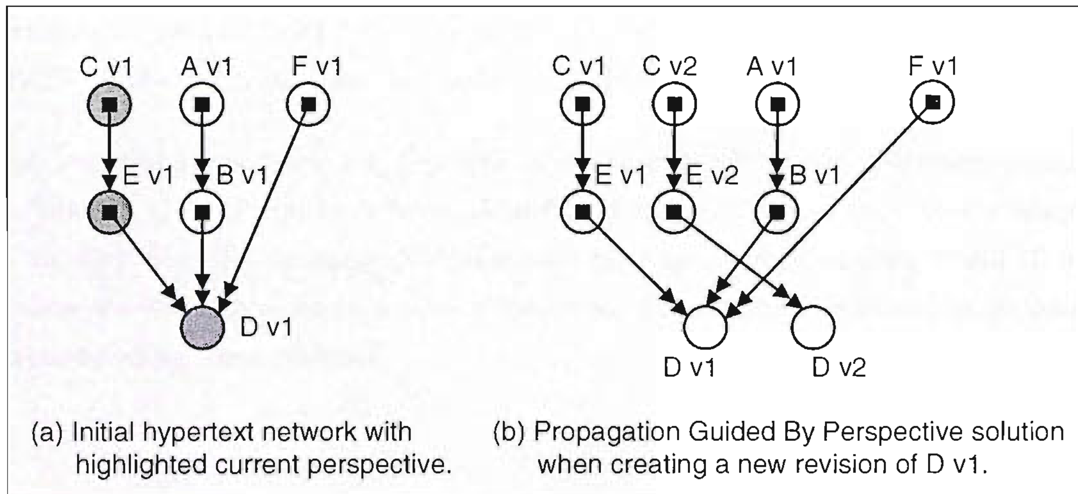


Figure 4.7: Propagation Guided By Perspective.

The solution by the NCM versioning model is to limit, but not eradicate, the amount of revision proliferation that takes place. This is by a technique called "Propagation Guided By Perspective". It allows users to restrict revision proliferation to only those User Context Nodes that belong to the perspective[5] through which the new revision was created. Figure 4.7(a) shows an example where the current perspective is 'D v1', 'E v1' and 'C v1'. Figure 4.7(b) shows the results of applying Propagation Guided By Perspective where only new revisions of 'E v1' and 'C v1' are created.

Section 10.6.2 illustrates the OCS data model solution which, unlike the NCM solution, offers the opportunity to eradicate the revision proliferation problem depicted by Figure 4.6.

4.7. Summary

This chapter has provided background to hypermedia versioning. This is to set the scene for the OCS data model's approach to versioning hypermedia structure (Chapters 9 and 10).

Selected within this chapter has been a range of hypermedia systems that offer different approaches to versioning. For example the use of a separate versioning server (CoVer, HVCF), systems that version nodes not links (HyperPro, HyperProp),

[5] This is the route chosen by the user from the top-most composite down to the node being manipulated.

systems that version links not nodes (Chimera), systems that version nodes and links (HVCF), and systems that version nodes embedded with links (DeltaV).

Also examined has been the problem of revision proliferation. Different systems (CoVer and HyperProp) have been identified that have devised their own strategies for dealing with the problem of unwanted object revisions. Chapters 9 and 10 also explain the OCS data model's own solution for the revision proliferation problems encountered by these systems.

Chapter 5. Referential Integrity Background

5.1. Introduction

This chapter explores the background to hypermedia referential integrity. This is the reliability of the references between objects of a hypermedia system. It is under investigation since a major benefit of the OCS data model is that it enhances the manner by which the referential integrity of hypermedia structure objects can be maintained (Chapter 11).

This chapter mainly concentrates on hypertext link referential integrity solutions for the navigational domain. This is because most hypermedia research has focused on this area since the majority of hypermedia systems specialise in this domain. Moreover, the general popularity of the Web has increased awareness of the need for link maintenance (of the navigational domain variety) even more so.

5.2. Importance of Referential Integrity

The role of hypermedia structure is to connect resources together. Hence referential integrity is a big issue as the success of hypermedia (as a concept and in practical application) is dependent on resource references pointing to the correct resource. Broken referential connections present a problem for three primary reasons [Ingham et al. 1996]:

1. Annoyance for the user attempting to follow the reference to the resource.
2. Results in a tarnished reputation for the reference supplier (i.e. the individual who offered the reference for traversal).
3. Possible lost opportunity for the owner of the resource pointed at by the reference.

5.3. Forms of Broken Hypermedia Structure

There are essentially four forms of broken hypermedia structure. These are dangling hypermedia structures, specious hypermedia structures, misaligned internal references, and broken internal routes.

Although this section is couched in terms of generic broken hypermedia structures each one can also be translated in respect of broken hypertext links of the navigational domain. This is useful because it is usually in terms of hypertext links that these hypermedia referential integrity problems are considered.

5.3.1. Dangling Hypermedia Structures

Dangling hypermedia structures occur when a hypermedia structure no longer references a valid resource [Davis 1995b; Ashman 2000a]. This may be caused by one of two reasons. The first is that the referenced resource has been deleted. The second is that the characteristics, upon which the hypermedia structure identifies the referenced resource, have been changed. For example a World Wide Web hyperlink may dangle if the destination document is moved to a new location and the link (embedded within the source document) is not updated to reflect the destination's new locality.

5.3.2. Specious Hypermedia Structures

Specious hypermedia structures occur when a different resource is moved to the same location as an existing resource. The result is that any hypermedia structure that referenced the original resource will now point at the wrong resource.

5.3.3. Misaligned Internal References

Misaligned internal references occur when a hypermedia structure points to the incorrect content within a given resource. This is also known as 'the content referencing problem' or 'the editing problem' [Davis 1998]. Misaligned internal references can occur when the referenced resource's content has been edited. For example a navigational hypertext link may point specifically to the second paragraph of a given document, but the content of that paragraph may later be moved to another area of the document. However if the anchor remains pointing to the second paragraph, the link's anchor will now be pointing at the wrong internal content.

In a sense misaligned internal references are an implicit form of specious hypermedia structure. This is because both types of hypermedia structure are broken as a result of them referencing the wrong location within a given resource. In the case of specious hypermedia structures, the broken structure is clearly pointing at the wrong internal content since it is referencing the wrong node resource. Therefore the distinction between the two types of broken hypermedia structure (specious and misaligned internal references) is based on the cause that leads to the structure breaking. For specious hypermedia structures, the cause of the break is because the entire content has moved to another location. For misaligned internal references, the cause of the break is because the referenced resource's content has been edited.

5.3.4. Broken Internal Routes

'Broken internal routes' (within a hypermedia connection) are a fourth lesser-known form of broken hypermedia structure. They occur when a hypermedia structure contains more than one internal route and one of those internal routes breaks in such a manner that its direct repair has an adverse effect on any other unbroken routes within the same hypermedia structure.

Figure 5.1 shows an example of a hypermedia structure containing more than one internal route. The first route expresses 'Trees Grows Leaves'. It is composed of objects N1, A1, E1, L1, E2, A2 and N2. The second route shows 'Trees Grow Acorns'. It is composed of objects N1, A1, E1, L1, E2, A2 and N3.

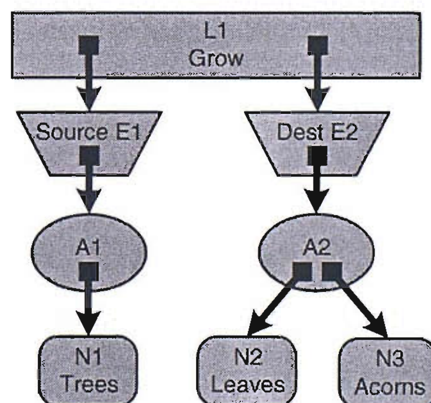


Figure 5.1: Hypertext link containing two internal routes.

Figure 5.1 can be used to provide an example of a broken internal route. For instance, the content of the 'Acorns' document may have been modified in such a way that

Anchor A2 now points to incorrect content within the 'Acorns' document. Thus the link now contains broken internal route N1 through to N3. This internal route is broken because it requires Anchor A2 to be updated. But the internal route N1 through to N2 is not broken as it does not require Anchor A2 to be updated. The process of and the implications arising from repairing the broken internal link of the example of Figure 5.1 form the subject of Section 6.5.

In the meantime, the problem of broken internal routes will not be discussed again in this chapter as little (if any) research has been conducted on them. But it is a problem that is directly relevant to the OCS data model, hence it is expanded upon in much greater detail in Section 6.5. Furthermore, Chapter 11 discusses the OCS data model solution to this problem.

5.4. Link Repair Strategies

This section lists some of the researched solutions into maintaining hypertext link integrity. [Davis 1998] and [Ashman 2000a] provide an exhaustive list of link maintenance solutions. What is clear is that there is no ultimate one-size-fits-all solution.

5.4.1 Passive Approach

This is the "Do Nothing" approach. Users making links to resources beyond their control (as is often the case when linking to World Wide Web documents) usually do so in good faith. It is not generally expected that resources will change significantly. Therefore it is hoped that links will not break very often.

5.4.2. User Onus

The most straightforward method for detecting and repairing broken hypertext links is to make it the responsibility of document owners to manually maintain their own links [Davis 1998]. Hence each document owner must continually check that their links reference a valid document (to avoid dangling links), that they reference the correct document (to avoid specious links), and that they point to the correct data within the referenced document content (to avoid anchor misalignment links).

The problem with this solution is that it places too much emphasis on user effort. The repetitive and time-consuming nature of these manual traversals almost certainly

means that users will not continue indefinitely with this link repair strategy [Fielding 1994].

5.4.3. Tightly Coupled Link Repair

Hypermedia systems that control access to both the links and nodes can enforce rigorous referential link integrity. This enables detection and repair of dangling, specious and anchor misalignment links.

Hyperbase Management Systems (HBMSs) are an example of an open hypermedia approach to tightly coupled link repair as all access to node data is conducted via the hyperbase. Therefore whenever a document is manipulated the HBMS can ensure that their links remain valid. However the problem with this approach is that any external tool wishing to access a document of the hyperbase will have to be adapted to negotiate this transfer with the hyperbase.

Tightly coupled systems have also been devised for the Web. The Atlas distributed hyperlink database system [Pitkow and Jones 1996] is one example. It works by assigning an Atlas Server to each Web server. The Atlas Server's role is to record a back link between all source and destination documents into a local database. When a destination document is modified, the Atlas system that records that destination document's back link, queries the local database for broken links. If found, that Atlas system then informs the source document's Atlas system to repair any links broken at the source end. Again the problem with this approach is that all the tools used for modifying documents must be adapted to communicate with the Atlas database.

5.4.4. System Tools for Pro-Active Users

Another range of link maintenance solutions is for the hypermedia system to provide tools that enable pro-active users to maintain the integrity of their hypertext links.

5.4.4.1. Forward References

Forward references [Ingham et al. 1996] are one of the simplest tools that Web page publishers have at their disposal. They correct dangling hyperlinks caused by documents moving to new locations. A forward reference consists of a dummy document being placed at the old document location which contains a link pointing to the document's new location. Forward references are only meant as a temporary solution whilst users update their hyperlinks to point to the new document location.

The problem is that it can never be known whether all users have updated their browsers to point to the new URL. This means it can never be known whether it is safe to remove the forward reference.

5.4.4.2. Link Integrity Checkers

Link integrity checkers can greatly automate the link integrity inspection process [Davis 1998]. Examples include spiders, robots and crawlers. Their primary purpose is for detecting and possibly repairing dangling links.

Web link integrity agents traverse the hypertext structure by retrieving documents to check their existence. They then recursively retrieve all other documents referenced by the checked documents to make sure that they exist too; and so the agent continues. If it is found that one or more documents are missing, the agent can either report this fact to the appropriate user (the method adopted by MOMSpider [Fielding 1994]) or attempt to perform link repair itself (the method preferred by the CLT/WW [Creech 1996]).

Link integrity agents are also employed for independently stored OHS links. These agents iterate over and test the links held at centralised link databases. They can also test for specious and anchor misalignment links if the link database records the last modification date of each document and the last modification date of the links that reference them. If the document was modified after the last time its links were modified, then the referenced document has either been swapped for a new document or its contents have been changed. Hence the link endpoints may no longer point to correct document content. The OHS can then warn the link holder of these potential dangers. Microcosm is an example of an OHS that implements such a scheme [Zhang et al. 2004].

The problem with link integrity checkers is that they are often reduced to little more than warning beacons. Either the agent cannot update links embedded in the document (since they are not its owner), or it does not know *how* to repair the broken link. For example if the document contents have been changed, then the link integrity agent will probably not know where to re-position the link endpoints within the modified document content.

5.4.5. Just-In-Time Link Repairs

Just-In-Time link repair schemes are so called, because they identify and repair anchor misalignment links just at the time when a document is about to be viewed [Vanzyl et al. 1994; Davis 1999]. They are used when links are stored separately from document content.

To carry out Just-In-Time link repair the link server must record a copy of the content at or around the anchor (of the link pointing at specific document content) at the time when the anchor is created. When the document is opened for viewing, the application can then examine the content pointed at by the anchor to check whether it is the same as that expected. If not, then a search algorithm can be executed in order to locate the correct content. The new anchor can then be written back to the link database.

The advantage with Just-In-Time link repairs is that documents may be edited by any application without needing to directly refer to the link service. But the disadvantage is that link repair can only be handled by the application, not the link server, since the representation of the offset of the anchor is opaque to the link service. This means that *all* viewing applications must be adapted to perform Just-In-Time link repair.

5.4.6. Preventing Broken Links Altogether

Some link maintenance solutions focus on preventing hypertext links from becoming broken in the first place.

5.4.6.1. Publishing Model

The Publishing Model permits hyperlinks to be made solely to read-only media [Vanzyl et al. 1994; Brush 2002]. This solves all dangling, specious and anchor misalignment problems as referenced documents cannot be deleted, moved or their contents changed. But it is an unrealistic general proposition to assume that documents will never change.

5.4.6.2. Hypertext Link Queries

Another option is to do away with the notion of static hypertext links altogether. Instead hypertext links can be expressed as queries in order to identify the document itself. Such link queries are commonly found on the Web used to create virtual

documents – documents compiled at runtime comprised of different document segments originating from different locations. This addresses dangling links caused by document movement. But if the destination document is deleted, then the link will continue to dangle.

5.4.6.3. Declarative Links

Declarative links [Davis 1999] do not represent link anchors as specific points in document content. Instead the user selects an area of document content and asks the system if there is more information about that particular content segment. An example of a declarative linking scheme is Microcosm generic links (Section 2.12.3). They match any occurrence of a given text string within any document.

The advantage with declarative links is that start anchors are relatively easy to implement. For example a user selection of text can be dispatched to the link service, which queries if any links exist that match this content that can be followed from the given context. But declarative end anchors are comparatively more difficult to implement since the scope over which documents to search for finding all occurrences of some given content may be left undefined. Thus the time to resolve such a query could be very large indeed.

5.4.6.4. Dynamic Links

Dynamic links [Ashman et al. 1997] are created at run-time as and when they are needed. Dynamic links are created by executing a link computation specification. This is a set of instructions that describe a range of attributes that if present within a document's content enable that document to participate within the dynamic link. Dynamic links are neither permanently stored in link databases or embedded within documents. Instead, the results of the dynamic computation are discarded after use. Therefore if the dynamic links are needed again, they must be re-computed.

An example of a useful application for dynamic links is a dictionary link where links can be formed between any occurrences of any English word to its dictionary definition. The text object is the source of the link, and the retrieved data is the destination. The dictionary link can be invoked on almost 100% of words within a given text document. Thus the dictionary link is most economically implemented as a dynamic link since the dictionary link type can potentially create a large number of links, most of which will never be used.

Dynamic links improve referential integrity due to them not being stored anywhere within the hypermedia system, this means that individual hypertext links do not need to be maintained. Consequently no dangling, specious or anchor misalignment links should occur. This is possible since dynamic links are not instantiated until the moment of use. At this point the computation (upon execution) can adapt its record of the links' source and/or destination references as a result of any changes to the data it is about to create links between.

However a major drawback with dynamic links is that they are computed. The problem being that hypertext link relationships are often a personal choice that reflects the personal associations made by an author. Thus it is typical that many links will not be able to be computed using a set of rules or algorithms. Hence dynamic links are often not a suitable method for authoring hypertext links.

5.4.7. Prevention via Versioning

Hypermedia versioning (Chapter 4) is another effective solution for maintaining the referential integrity of hypertext links [Østerbye 1992; Vitali 1999]. It is the most important referential integrity solution as regards the OCS data model since a key benefit of the OCS data model is that it improves hypermedia versioning (Chapters 9 and 10).

5.4.7.1. Versioning Hypermedia Documents

Versioning just hypermedia documents can prevent all three link maintenance problems. It solves the problem of anchor misalignment links since document content cannot be changed. Therefore all existing link endpoints will continue to point at the same anchor hotspots. It prevents dangling links because versioned documents cannot be deleted. And it prevents specious links because versioned documents cannot be swapped for other documents in their place.

However versioning does bring disadvantages. Firstly, there will be many more documents and links to manage. Not only will new revisions of documents be created whenever a document is modified, but new copies of links will also be created to reference the new document revisions in order to maintain the same inter-document relationships as the original document revision. And secondly there is extra overhead when links are stored separately. This takes the form of an additional attribute that must be appended to each link anchor in order to identify to which revision of the document a given link belongs [Davis 1998].

5.4.7.2. Versioning Hypertext Links

Versioning hypertext links can help referential integrity when links are stored separately from document content. This is because there is a danger that changing the content of the individual hypermedia objects that make up a hypertext link (or even deleting them) can cause link breakage. For example dangling links can arise if one or more of the objects within a link are deleted, or if the document location details stored by the link's hypermedia objects (e.g. OHP-Nav Node or FOHM Data objects) are changed. There is also a danger of anchor misalignment links occurring if the anchor identification content recorded within a link is amended to point to a different area within a document when no document content has actually been changed.

Such link breakages can be prevented if links are versioned whereby any alterations to a link lead to the existing link being preserved and a new revision of the link being created containing the new linking data. But hyperlink versioning cannot correct specious links. If a document has been moved to the same location as an existing document, then the versioned link will still point to the newly positioned document rather than the original at the new location.

5.4.7.3. Versioning Hypermedia Documents and Hypertext Links

Versioning both hypermedia documents and separated hypertext links together can further improve the referential integrity of links between documents.

The danger with versioning documents alone is that the existing unfrozen links (that reference the original document) may be overwritten with new reference data so that they point at the new document revisions instead. Thus the original frozen document revisions may no longer be linked. If the hypertext links were versioned too, then new revisions of the existing links referencing the original document would be created at the same time as a new revision of the document was created. Hence both original document and referencing links would be preserved.

The disadvantage with this combined approach is that it combines the disadvantages of both separate versioning schemes, i.e. many more documents and links will probably end up having to be managed. Thus consideration has to be given as to which document and link changes should lead to new revisions as, like with the separate schemes, it is unreasonable to expect every single document and link change to be recorded.

5.5. Summary

The focus of this chapter has been existing research on hypermedia referential integrity. This is to aid understanding of how the OCS data model can improve the referential integrity of hypermedia connections (explained in Chapter 11).

Three forms of broken hypermedia structure are the most prevalent: dangling, specious and misaligned internal references. The most and current research centres on hypertext link maintenance for the navigational domain. Therefore these research solutions have been the main preoccupation of this chapter.

What is evident is that there is no ultimate solution to the broken link problem. Instead a range of solutions has been developed. They include:

- Do nothing and hope that the links won't break.
- Assume that the users, who will be the victim of any broken links, are sufficiently meticulous to routinely check their links for breakages.
- Use tightly coupled link repair schemes. These control access to both links and nodes which enables rigorous referential link integrity to be enforced. Examples include HBMSs.
- Encourage the owners of the documents that have caused links to break to adopt a pro-active approach. This can range from forward references to link integrity checkers.
- Perform Just-In-Time link repair whereby broken link detection and/or link repair is carried out just at the moment when the link is about to be called into use.
- Adopt measures that prevent hypertext links from breaking in the first place. This can include adopting the Publishing Model, representing hypertext links as queries, or using declarative links.
- Version all hypermedia documents and/or hypermedia links.

It is the last solution that is of most interest since the OCS data model benefits and provides an environment for hypermedia structure versioning. Thus the OCS data model widens the scope for using versioning as part of a referential integrity solution for hypermedia structural resources.

Chapter 6. Problem Domain Issues

6.1. Introduction

As reported in Chapter 3 it was in developing the Solent CB-OHS and experimentation with versioning FOHM structures that inspired conception of the OCS data model. Four problematic issues arose as a result of this work:

PD1:	The use and storage of hypermedia objects that carry out the same function.

PD2:	The use and storage of identical hypermedia structure.

PD3:	The dangers of revision proliferation.

PD4:	The confusion caused by repairing broken internal routes.

This chapter details the exact nature of these problems, for it was to address these issues that led to the development of the OCS data model.

6.2. Issue One: Repetitive Hypermedia Objects

Experimentation with the Solent CB-OHS and Auld Linky revealed that many of the hypermedia objects that belonged to different hypermedia structures actually fulfilled the same role as one another.

6.2.1. Repetitive Hypermedia Objects in OHP-Nav

Figure 6.1 displays three OHP-Nav hypertext links, many of whose hypermedia objects perform the same role. Such objects are indicated by being connected together by thick grey lines with circles at each end. It is even possible that some 'identical' objects exist within the same hypermedia structure, an example of which is indicated

by Endpoints E1 and E2 both specifying a bi-directional value within the hypertext link denoted by L1.

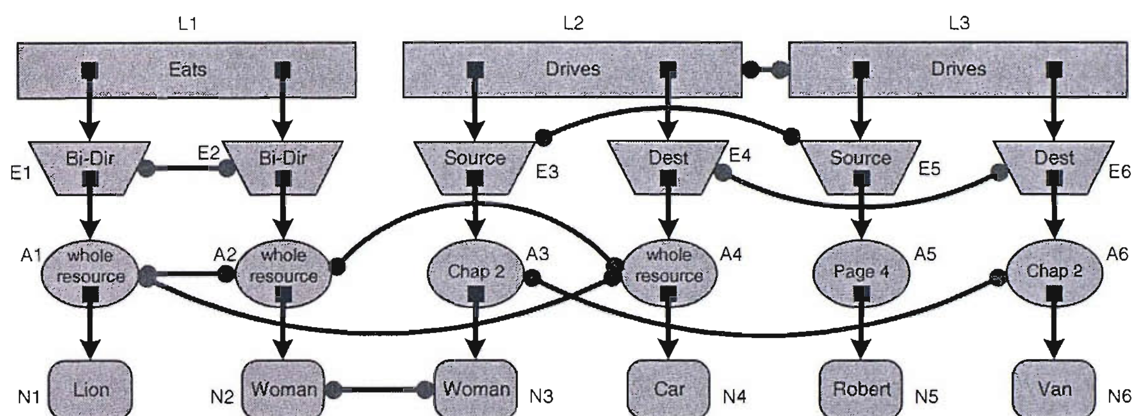


Figure 6.1: OHP-Nav hyperstructures containing 'identical' hypermedia objects.

As a reminder to the reader, Table 6.1 pinpoints which hypermedia objects of Figure 6.1 carry out the same function. Section 3.2.3 describes the roles performed by each of the individual hypermedia objects that make up OHP-Nav hypertext links.

Object	Function
Link objects L2 and L3	Describe the same link type (Drives).
Endpoints E3 and E5	Specify the same direction (Source).
Endpoints E4 and E6	Specify the same direction (Destination).
Endpoints E1 and E2	Specify the same direction (Bi-Dir).
Anchors A1, A2 and A4	Point to the whole resource.
Anchors A3 and A6	Point to the same hotspot (Chapter 2).
Nodes N2 and N3	Point to the same document (Woman).

Table 6.1: Hypermedia objects of Figure 6.1 performing the same function.

Because the objects of Table 6.1 appear identical in their functionality, it would seem sensible to be able to re-use a single copy of each object where that object's functionality is duplicated elsewhere. For example use a single 'Drives' Link object in place of L2 and L3; or use a single 'whole resource' Anchor object in place of Anchors A1, A2 and A4.

However, despite some of those objects seemingly sharing the same role, they cannot be considered identical due to the manner by which OHP-Nav organises structure. This is because each object is embedded with the different identifiers of the objects to which they are associated in the linked-list fashion described in Section 3.2.5. For

example Anchor A1 is embedded with Node N1's object identifier; Anchor A2 is embedded with Node N2's object identifier; and Anchor A3 is embedded with Node N3's object identifier.

The result is that neither Link objects, Endpoints or Anchors can be re-used as independent entities. This is for two reasons.

1. Single functionally identical objects may have different content because they are embedded with different object reference identifiers within their content. Hence one functionally identical object cannot be directly swapped for another. The two hypermedia objects L2 and L3 provide a good example of this problem. They are both functionally identical since they both describe a 'Drives' relationship. But they actually have different content from one another as each Link object is embedded with different connection data. E.g. L2 is embedded with connection data pointing to E3 and E4; and L3 is embedded with connection data pointing to E5 and E6. Therefore even though both L2 and L3 are functionally the same, they cannot be swapped with one another because they are still physically different from one another due to embedded content.

2. The linked-list nature of object reference embedding means that each object type can only be re-used in association with the objects it references. A Link object can only be re-used with the Endpoints it references; Endpoints can only be re-used with the Anchors they reference; and Anchors can only be re-used with the Nodes they reference. Hence neither object type can be used as a single object. For example if wanting to re-use Endpoint E1 (of Figure 6.1) then it can only be re-used in conjunction with Anchor A1, and Anchor A1 can only be re-used in conjunction with Node N1.

The only OHP-Nav object types that can be re-used as independent entities are Nodes. This is because they do not contain an embedded reference to another object type. Figure 6.1 contains an example of two Nodes (N2 and N3) that are functionally identical. Hence the two structures denoted by Link objects L1 and L2 can be re-organised to share a single copy of the 'Woman' Node. This new arrangement is shown in Figure 6.2. Node N3 has been eliminated, and Anchors A2 and A3 now jointly reference N2.

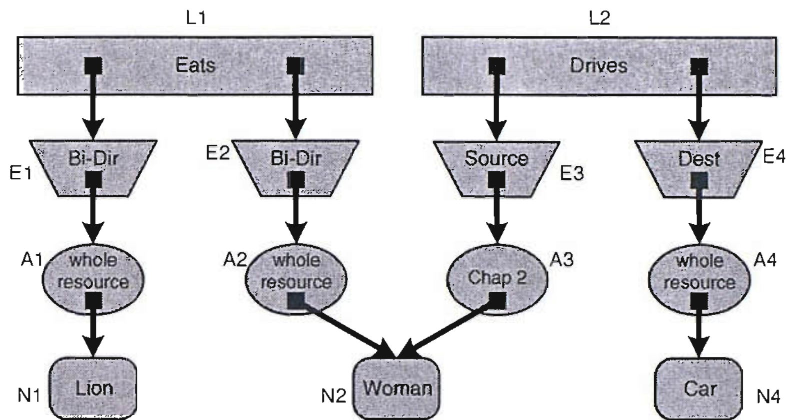


Figure 6.2: Node re-use in OHP-Nav.

As demonstrated, the opportunity for individual object re-use within the OHP-Nav data model is very limited. For example Table 6.1 shows that there is scope for re-using 7 individual objects within the three hypertext links of Figure 6.1. This would eliminate the need for 8 (repetitive) objects. But all the OHP-Nav data model can allow is the re-use of one object, N2, which has meant that only one other object, N3, can be deleted.

6.2.2. Repetitive Hypermedia Objects in FOHM

Re-use of individual objects within FOHM is also largely ruled out. This is due to either the wholesale embedding of one FOHM object within another, or the embedding of object references within FOHM objects. The reader is reminded of the roles performed by each of the individual objects that make up FOHM structure by referring to Section 3.3.4.

Object embedding prevents re-use of the embedded object as a single object since its re-use also necessitates re-use of the containing object as well. This particularly applies to Binding objects as they are always embedded within an Association object. Thus they can never be re-used as single objects. Embedded Bindings also means that Associations cannot be re-used as single objects since all embedded Bindings within an Association will automatically be re-used along with the Association itself. Furthermore, Reference objects cannot be re-used as single objects as any embedded Data or Association objects (within the Reference object), or Data or Association objects pointed at by the Reference object's embedded references, will be re-used too. Likewise, Data objects that are already embedded within Reference objects cannot be

re-used as single objects as they cannot be re-used without the containing Reference object.

However there is one exception. Data objects which are not embedded in Reference objects can be re-used as single objects since FOHM Data objects do not contain embedded references to other object types nor do they contain embedded objects.

Figure 6.3[6] provides an example of the difficulty of re-using single FOHM objects which appear to contain identical functionality.

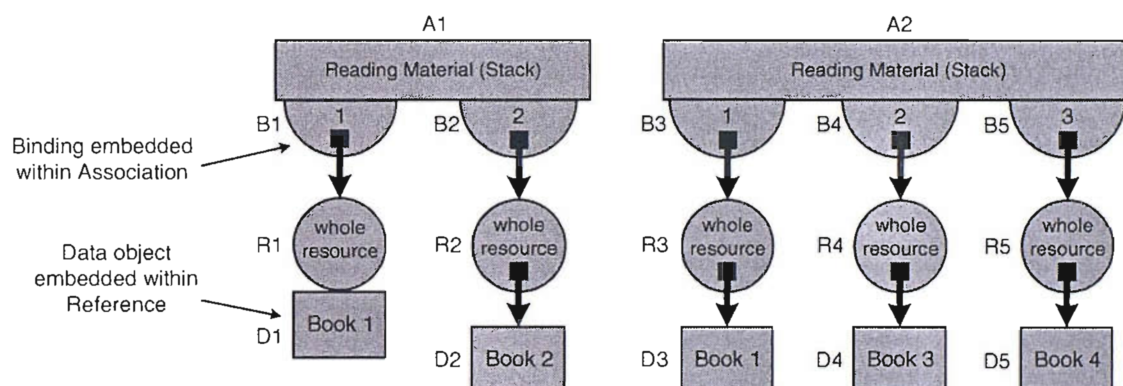


Figure 6.3: Two FOHM hyperstructures containing 'identical' hypermedia objects.

Three examples of single object re-use failure within Figure 6.3 are noticeable:

1. Despite Bindings B1 and B2 being functionally identical to B3 and B4, each Binding cannot be re-used in place of the other because the Bindings are wholly contained within their associated Associations. Hence each Association (A1 and A2) would have to be re-used too.

2. References R1 through R5 can also be considered functionally identical since they all describe 'whole resource' References. But they still cannot be re-used in place of one another. R1 cannot be re-used, because it contains embedded object D1. Hence R1 cannot be re-used without also re-using D1, the problem being that R2 to R5 do not contain any embedded Data objects. Furthermore, objects R2 through R5 cannot be re-used in place of one another as single objects because in each case it would mean re-using the objects they reference, i.e. objects D2 through to D5.

[6] One object being embedded within another is shown by the two objects touching each other such that the top object is the containing object and the object immediately below is the contained object.

3. Even though Data objects D1 and D3 are functionally identical (i.e. they describe the same content), they still cannot be re-used in place of another. This is because they are structurally different - one is an embedded object whilst the other is a referenced object. As a consequence Data objects D1 and D3 are actually stored twice in their different manifestations. The data within D1 is stored as embedded content within object R1 since D1 is embedded within R1; and object D3 (not embedded within a Reference object) is stored as an individual object.

Once again, as was the case with OHP-Nav, the opportunity for individual object re-use is severely limited. This time it is due to object reference embedding combined with the practice of embedding entire FOHM objects within one another.

6.2.3. Dynamic Linking

Dynamic linking (Section 5.4.6.4) offers a potential solution to the problems posed by embedded object references.

Generally they are used within the field of navigational hypermedia to connect document node resources together. As described in Section 5.4.6.4 dynamic links rely on computation specifications, executed at run-time, to identify which documents are to be connected together.

The reason why dynamic links can help with the problem of hypermedia object reference embedding is because these types of links are not stored anywhere within the hypermedia system. Instead, they are dynamically created and discarded after use. Therefore if there are no hypertext links within the system, then there are no hypermedia objects. If there are no hypermedia objects, then there are no embedded object references. Thus with the removal of permanent hypertext links within the hypermedia environment, there is no longer a hypermedia object organisation problem that needs to be solved through adopting the OCS data model.

However, as also explained in Section 5.4.6.4, the use of dynamic links to connect all documents together within the hypermedia environment is neither practical nor realistic. This is because link relationships are often a personal choice that reflect the personal associations made by an author. In most cases such personalised links cannot be computed based on a set of rules. Consequently most hypertext links remain hand-made and the problem of embedded hypermedia object references remains.

6.2.4. Repetitive Hypermedia Object Problem

There are two reasons why repetitive hypermedia objects within hypermedia structure are flagged as a problematical issue.

1. *Storage Space.* It is a waste of resources to store duplicate objects. Moreover the greater the number of objects in storage, then the more objects there are to search through. Thus it will take longer to locate relevant objects. If multiple objects are being dispatched across a network, but more than one are 'identical', then more objects will be sent than is really necessary.

2. *Open Hypermedia.* Embedded hypermedia object references mean that single hypermedia objects are not open to re-use. This is because each object can only be used in conjunction with the objects to which it is physically attached.

6.3. Issue Two: Repetitive Hypermedia Structure

When creating structures with the Solent CB-OHS and Auld Linky it was often observed that the segments of one structure were found repeated within other independent structures. Therefore it seems intuitive that such repetitive structure segments should be able to be re-used within the same or between different structures.

6.3.1. Repetitive Hypermedia Structures in OHP-Nav

Figure 6.4 shows two OHP-Nav hypermedia structures. The only differences between them are the destination Nodes: Hyperlink A has 'Meat' as its destination, whilst Hyperlink B has 'Fish' as its destination.

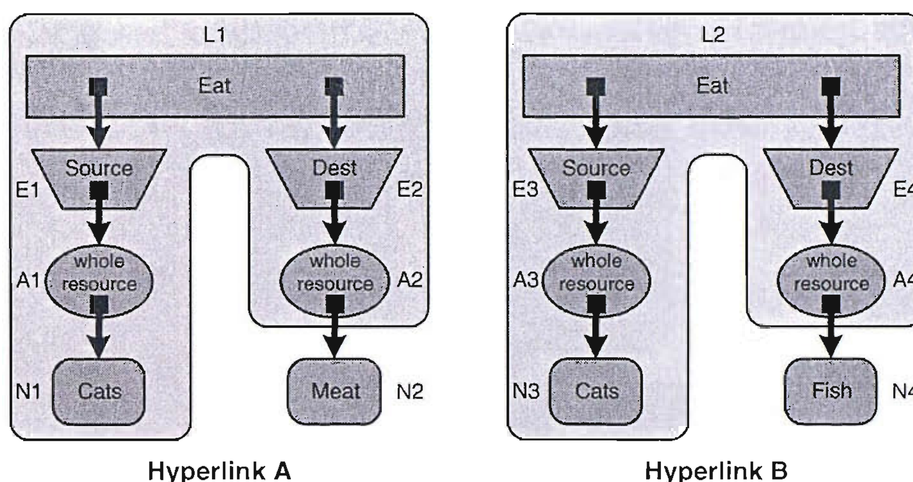


Figure 6.4: Example of a repetitive segment in two hypermedia structures.

Both hypertext links contain functionally identical segments that represent the same relationship, namely that 'Cats Eat' something. This common relationship is indicated by both segments being highlighted within Figure 6.4. It therefore seems logical to re-use the highlighted segment of Hyperlink A in place of the highlighted segment of Hyperlink B such that connected objects N1 to A2 replace N3 to A4. But a direct swap is not possible due to OHP-Nav embedded object references for two reasons.

The first reason is because Anchors contain embedded references to Nodes. This means that in order to make A2 reference N4, then A2's content (i.e. its embedded reference) needs changing. But making such changes will prevent that same structure from being re-used at its original location. For example if Hyperlink A's re-used segment was changed to reference N4, then when the segment is once again used in relation with Hyperlink A, the segment's destination will now point to 'Fish' Node N4 instead of 'Meat' Node N2.

The second reason is because the re-use of Hyperlink A's highlighted segment would also necessitate the re-use of Node N2. This is because A2 contains an embedded object reference that points directly at N2. The problem is that N2 is not meant to be part of the re-used segment. Hence this particular highlighted segment cannot be re-used on its own. If re-use were to proceed anyway, then the wrong structural organisation would result, namely that 'Cats Eat Meat and Fish' (Figure 6.5). This is not the intended structural organisation for Hyperlinks A and B as the two relationships are meant to be recorded as separate structures.

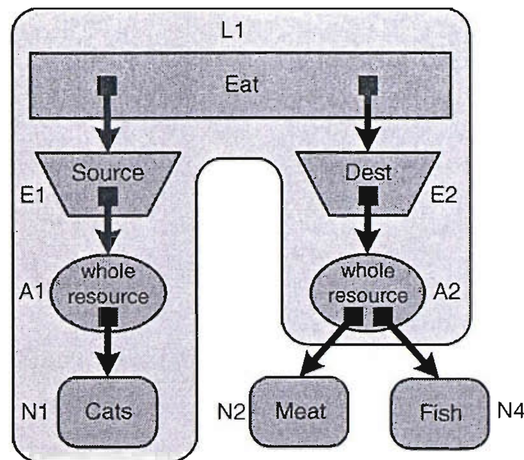


Figure 6.5: The wrong structural organisation: 'Cats Eat Meat and Fish'.

However, OHP-Nav does enable a limited form of structure segment re-use. As indicated by the 'Cats Eat' relationship example above, any structure segment wanting to be re-used must pass two tests:

1. No segment objects must contain references to objects outside of the re-used segment.
2. The segment objects must not be changed in order to be re-used.

The result is that all re-used structure segments will comprise at least one Node object since Nodes are the only object type that do not contain embedded references to other objects. The only Anchors permitted within re-used structure segments will be those whose embedded references point at existing Nodes within the structure segment. This is to ensure that rule 1 is never violated. Likewise, if the structure segment also contains Endpoints, then their embedded references must also point at existing Anchors of the structure segment. And if the structure segment contains any Link objects, then their embedded references must also point at existing Endpoints within the structure segment.

With these restrictions in mind, OHP-Nav only allows a limited number of re-usable structure segments to be created within Hyperlink A. They are:

- N1 on its own.
- N1 and A1.
- N1, A1 and E1.

- N2 on its own.
- N2 and A2.
- N2, A2 and E2.
- N1, A1, E1, L1, E2, A2 and N2.

However setting OHP-Nav's limitations aside, there are many more potentially re-usable structure segments that exist within Hyperlink A. This includes a Node-less 'Eat' relationship comprising A1, E1, L1, E2 and A2. Such a relationship could be applied to an altogether different set of Nodes, for example to create 'Birds Eat Worms'. Thus there exists a Boolean algebra of possible arrangements, including things that 'Eat Meat' and 'Eat Fish'. But OHP-Nav does not permit such arrangements because A1 and A2 are embedded with references to Nodes N1 and N2 which would have to be re-used as well.

6.3.2. Repetitive Hypermedia Structures in FOHM

The opportunity for repetitive hypermedia structure segments appearing within FOHM structures is even more striking. This is because the same substructure segment may be repeated more than once within the same hypermedia structure itself. Such a situation is possible due to FOHM Reference objects being able to reference Association objects. The result is a potential for very large structures. Figure 6.6 shows an example.

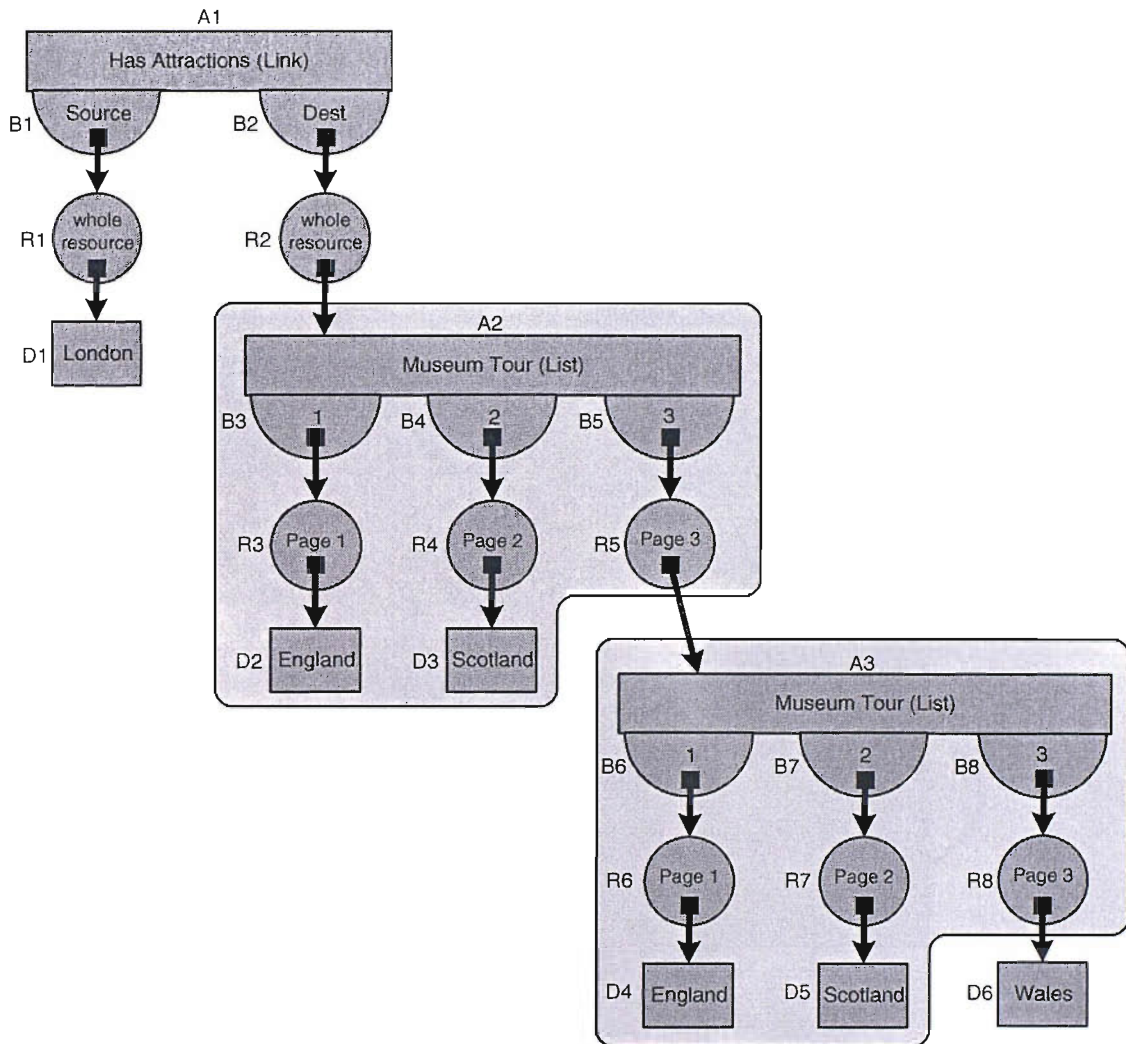


Figure 6.6: FOHM hyperstructure containing repetitive structure.

Figure 6.6 shows two highlighted segments which are structurally identical as all objects within both segments perform the same roles and are connected to one another in the same fashion. Yet they are not considered the same. This is because each object is embedded with the different identifiers of the objects to which they are attached (in a linked list fashion). For example Binding B3 is embedded with object identifier R3, whilst the seemingly identical object Binding B6 is embedded with object identifier R6. Consequently the two hypermedia substructure segments are two separate entities consisting of separate objects and connections.

The reason why the two segments cannot be re-used in place of one another is due to the connections that the repeated substructures have with objects external to the repeated segments. If the first segment were re-used at the position of the second segment, then the incorrect structural formation of Figure 6.7 would result.

Of note is the confusing role of object R5 which re-attaches itself back into the re-used segment. The original role of R5 (when the two segments are separate) is to point to object A3 of the second segment. But segment re-use means that A2 now replaces A3. Thus the new role of object R5 is to point to A2 (as well as taking on the role of R8 which points to D6). Thus the overall result is an erroneous loop within the structure which produces the wrong structural formation.

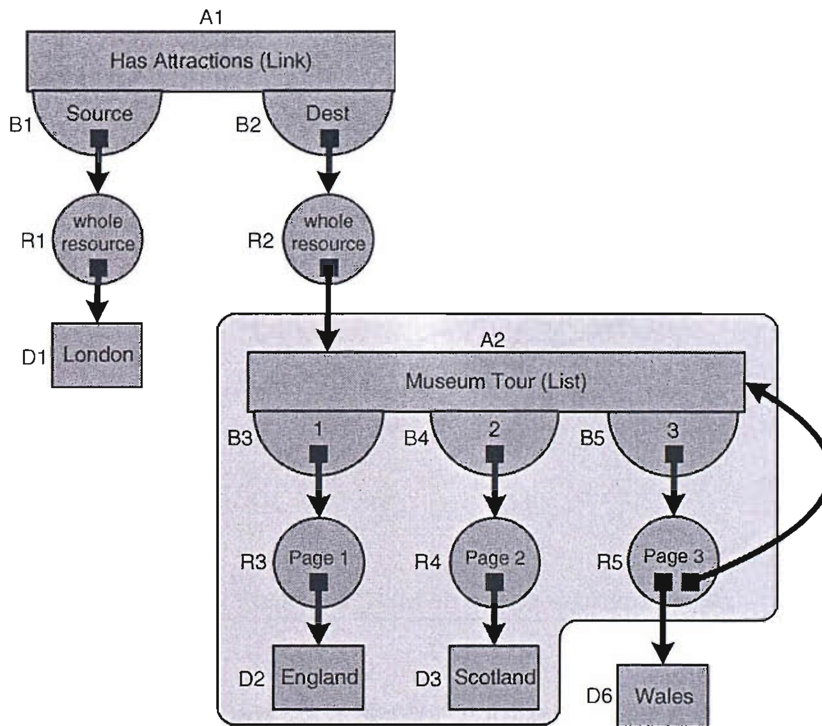


Figure 6.7: Incorrect re-use of hyperstructure segment.

However, like OHP-Nav, FOHM does offer limited re-use capability which enables certain structure segments to be re-used. For example the original FOHM hyperstructure of Figure 6.6 could be re-organised to incorporate the re-used structure segments comprising R3, R4, D2 and D3. This is shown in Figure 6.8. Such re-use is possible because FOHM References can be referenced by multiple Binding objects. Hence Binding B6 can connect to Reference R3, and Binding B7 can connect to Reference R4.

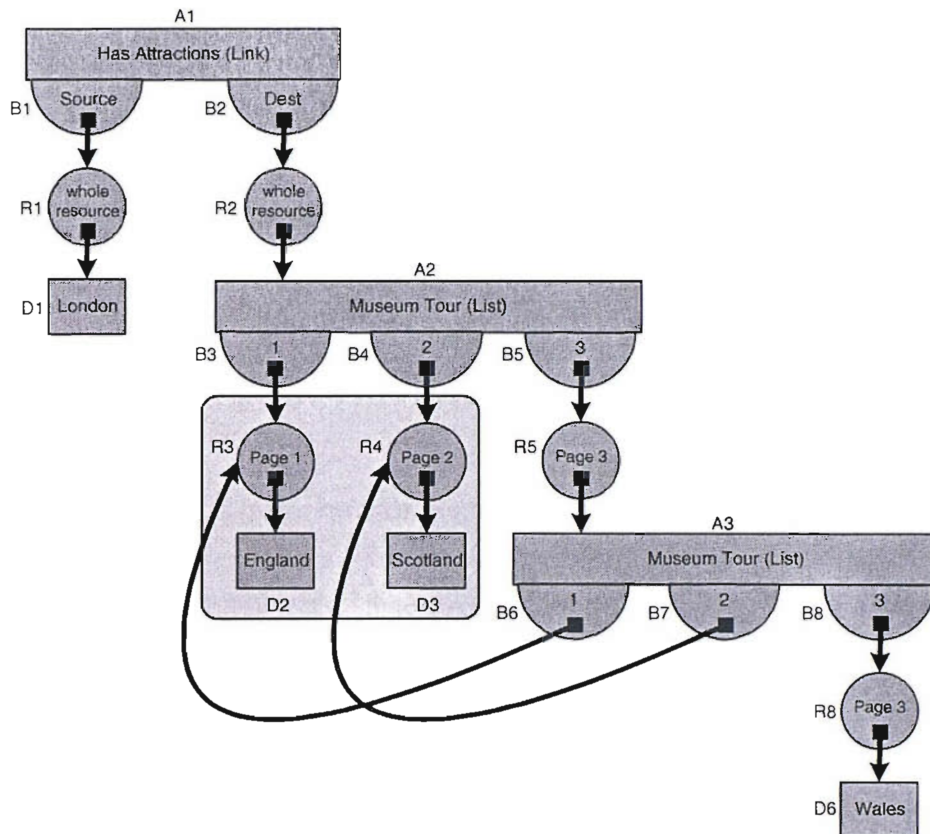


Figure 6.8: Possible FOHM structure segment re-use.

However this solution is not as efficient compared with re-using the original highlighted segment of Figure 6.6 since objects A3, B6, B7, B8 and R8 are still not being re-used. Moreover the structure designer may actually want the structural layout of Figure 6.6 since it provides a clearer representation of the structure. I.e. the overall hypermedia structure is effectively divided into three separate segments which may be considered easier to create and maintain compared with the more complicated re-use scheme of Figure 6.8.

6.3.3. Repetitive Hypermedia Structure Problem

The inability to re-use hyperstructure segments is identified as a problem issue for two reasons.

1. *Storage Space*. It is a waste of resources to duplicate objects and connection data. As with Issue 1, the more objects in storage, then the greater the number of objects there are to search through. This means it takes longer to locate relevant structural formations of objects. Also, if the same structural formations exist within a single hypermedia structure that is being dispatched across a network,

but each segment is duplicated as a separate segment, then more data will be dispatched than is really necessary.

2. *Open Hypermedia*. Again, this situation violates the notion of open hypermedia as whole segments of hypermedia structure cannot be re-used since structure can only be used in relation to the specific objects that it connects.

6.4. Issue Three: Revision Proliferation

Versioning hypermedia structure offers real benefits as described in Chapter 4. Consequently experimentations took place in versioning both OHP-Nav and FOHM structure. Versioning OHP-Nav structures resulted in creation of the OHP-Version protocol (Section 6.4.1); and experimentation with versioning FOHM structure led to the Contextualised Connections approach (Section 6.4.2). However both approaches encountered the very real problem of revision proliferation (also described in Chapter 4).

6.4.1. The OHP-Version Protocol

The purpose of the OHP-Version protocol [Griffiths et al. 2002] is to enable versioning of all OHP-Nav hypermedia objects. It provides creation, deletion, update and retrieval operations for all hypermedia objects. Its primary mandate is to create a new revision of an object whenever an attempt is made to update an existing object. For example changing the contents of an Anchor object so that it points to a different internal location within a given resource will result in creation of a new revision of that Anchor object. The original Anchor object will persist with original content, whilst the new Anchor revision will contain the updated content.

But as already mentioned a key problem faced by OHP-Version is revision proliferation. When a new revision of an OHP-Nav hypermedia object is created it can lead to the creation of more objects than just the expected new revision. This is due to the embedded linked list formation between OHP-Nav hypermedia objects.

Figure 6.9 provides an OHP-Nav example. The user wants to write new content to Anchor object 'A2 v1' of Hyperstructure A. This is meant to result in the formation of Hyperstructure B, where new Anchor object 'A2 v2' is simply appended to the hypermedia structure as shown.

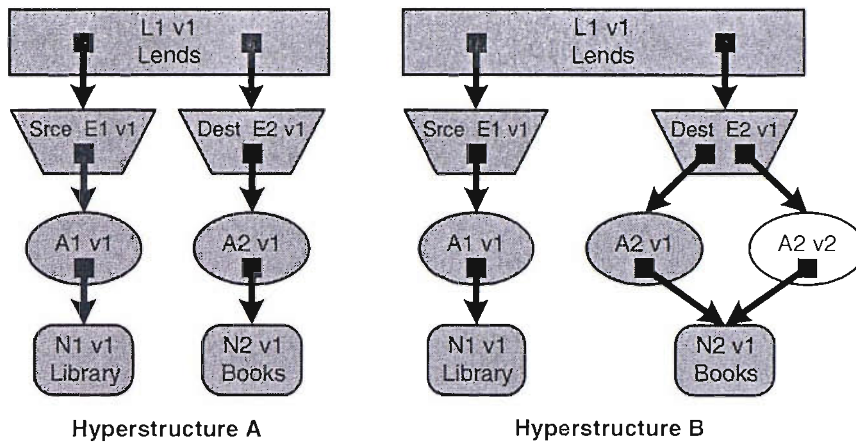


Figure 6.9: Initial and expected OHP-Nav hypermedia structures.

However the hyperstructure will not turn out as intended. Instead it takes the formation displayed in Figure 6.10. This is because OHP-Nav causes revision proliferation to take place where, as can be seen, more new object revisions have been created than just the one expected new revision.

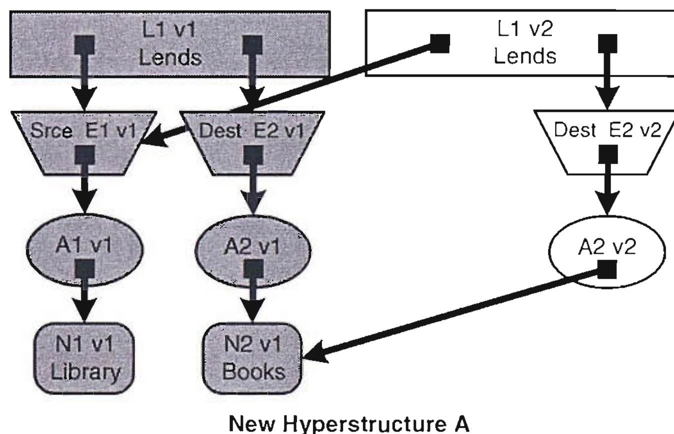


Figure 6.10: Resulting OHP-Nav hyperstructure after OHP-Version application.

The problem lies with OHP-Nav embedding object references within hypermedia objects whereby object references become part of an object's internal content. E.g. Node object identifiers form part of an Anchor object's content; Anchor object identifiers form part of an Endpoint object's content; and so on (Section 3.2.5).

This has consequences when attempting to tie the new revision into existing Hyperstructure A of Figure 6.9. That is, attaching Anchor 'A2 v2' to Node 'N2 v1' and Endpoint 'E2 v1'.

Difficulties arise because the object identifiers required to reference the new revision need updating. In the case of the example, this means updating object 'E2 v1' since it is Endpoint objects that are responsible for storing Anchor object identifiers. (Remember that Nodes do not need store Anchor object identifiers, therefore Node 'N2 v1' does not need updating.)

But it is this act of updating the object reference identifiers located within other objects that will, in turn, cause a new revision of that object (containing the object reference identifier) to be created too (since its content, i.e. the object reference identifiers, is being modified). Once again, attempting to tie this object back into the original hypermedia structure may lead to further revision proliferation since it may also be necessary to update more object reference identifiers found in other objects. This revision proliferation will continue until there are no more references within other hypermedia objects that need updating. Indeed, the only time that revision proliferation does not occur is when updating a Link object. This is because no other object type stores references to them. Hence when a Link object is updated, only the one new revision of that object is created.

Figure 6.10 shows the revision proliferation that occurs as a result of attempting to create a new revision of object 'A2 v1'. Instead of just one new object revision being created a further two objects are also created. The steps that lead to the hyperstructure of Figure 6.10 are listed in Table 6.2.

Step	Action
1	New revision 'A2 v2' is created.
2	Attempting to connect 'A2 v2' to 'N2 v1' succeeds. This is because Node objects do not store the embedded object reference between Nodes and Anchors. This is the Anchor's responsibility. Hence the Node's contents are not updated which means that no new revision of the Node needs to be created.
3	Attempting to connect 'A2 v2' to 'E2 v1' causes 'E2 v1' to be modified as Endpoints store the connections between Endpoints and Anchors. Hence 'E2 v2' is created.
4	Attempting to connect 'E2 v2' to 'L1 v1' causes 'L1 v1' to be modified as Link objects store the connections between Link objects and Endpoints. Hence 'L1 v2' is created.
5	Attempting to connect 'L1 v2' to 'E1 v1' succeeds. This is because Endpoints do not store the embedded reference between Link objects and Endpoints. This is the Link object's responsibility. Hence the Endpoint's contents are not updated which means no new revision of the Endpoint need to be created.

Table 6.2: Revision proliferation caused by creating new revision 'A2 v2'.

This is a far from ideal situation. Many more objects have been created than expected. In this case the two additional objects 'E2 v2' and 'L1 v2'. These are duplicates of the original objects. The only difference between the new and original revisions is that the new revisions point to different hypermedia objects (i.e. they are embedded with different hypermedia object reference identifiers). This can cause great confusion for clients as the new object revisions are not only unexpected and unintended, but are seemingly pointless additions to the hyperstructure. Creating new objects just to store updated object references is simply not good versioning practice.

Even the intended final hypermedia structure formation as shown by Hyperstructure B in Figure 6.9 is confusing. What does this formation represent? Does it represent the new revision of the hyperstructure formation whereby the latest hyperstructure revision is meant to comprise a total of three Anchor objects? Or is the new hyperstructure formation simply a container for two different revisions of hypermedia structure that are recorded within the same hyperstructure?

For the second scenario it would mean that the first hyperstructure revision would comprise Anchor 'A2 v1', but not Anchor 'A2 v2'. And the second hyperstructure revision would comprise Anchor 'A2 v2', but not Anchor 'A2 v1'.

The OHP-Version protocol adopts the position of the second scenario whereby a versioned hypermedia structure acts as a container for different revisions of hypermedia structure. It uses Configuration Objects that record which object revisions are members of which hypermedia structure revision. For example Hyperstructure B could be represented by two Configuration Objects:

- Configuration A would record the early hyperstructure revision as comprising: 'N1 v1', 'A1 v1', 'E1 v1', 'L1 v1', 'E2 v1', 'A2 v1' and 'N2 v1';
- Configuration B would record the later hyperstructure revision as being: 'N1 v1', 'A1 v1', 'E1 v1', 'L1 v1', 'E2 v1', 'A2 v2' and 'N2 v1'.

But at the end of the day the OHP-Version protocol is still confronted with the problem of revision proliferation. This is due to the conventional object reference embedding approach of the OHP-Nav protocol.

6.4.2. Contextualised Connections

A different method was adopted when experimenting with versioning FOHM structures. This was called Contextualised Connections (also known as FOHM versioning) [Griffiths et al. 2002]. It reduces revision proliferation, but does not completely eradicate it. Like the OHP-Version protocol, the Contextualised Connection approach provides creation, deletion, update and retrieval operations for all FOHM hypermedia objects. It also creates new revisions of objects whenever attempts are made to update existing objects. However, the Contextualised Connections approach is different from OHP-Version, because it draws on virtual objects within FOHM structure to assist with revision selection.

6.4.2.1. Adjustments to the FOHM Data Model

Before FOHM versioning can take place certain adjustments need to be made to the basic composition of FOHM structure:

1. Binding objects need to be promoted to become first class objects so that they are no longer enclosed within Association objects. This enables Bindings to be versioned separately from Associations.

2. The FOHM data model needs updating to allow multiple connections between objects. This enables a FOHM object to be connected to multiple object revisions.

Figure 6.11 shows an example of the new structural arrangements that FOHM structure can undertake.

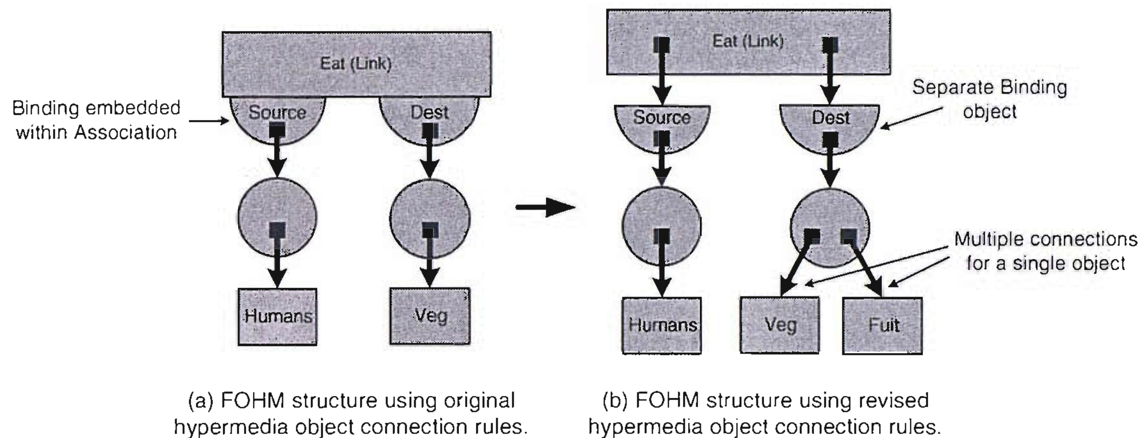


Figure 6.11: Example of FOHM structure organisation before and after it was revised to enable versioning.

6.4.2.2. Contextualising FOHM Connections

FOHM objects continue to store embedded object references that point at the object identifiers of one or more other FOHM objects as described in Section 3.3.4. However for each object that references a versioned object, instead of connecting directly to the versioned object, it instead stores a search pattern. This search pattern effectively acts as a pointer to a virtual object, i.e. an object that does not physically exist. The virtual object's role is to act as an informal pointer to a set of related revisions. Figure 6.12 shows an example where Binding B1 contains an embedded object reference to Reference R1 (a virtual object) where R1 represents a set of related revisions.

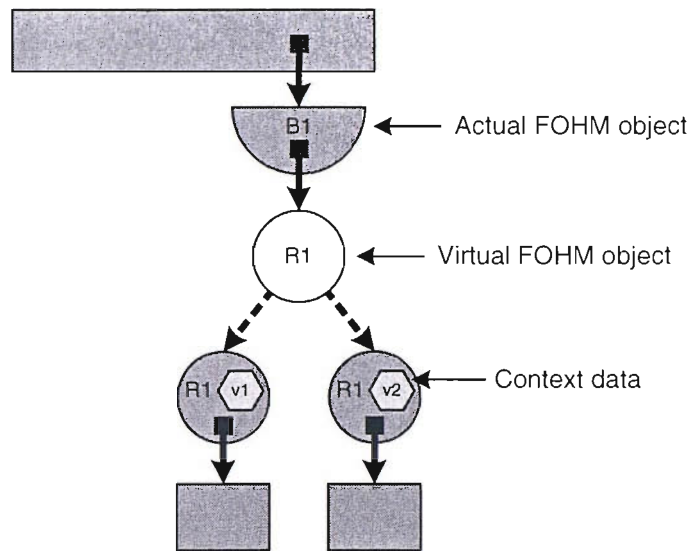


Figure 6.12: Example of a contextualised connection.

Which object revision is selected (either 'R1 v1' or 'R1 v2') is determined by context. Every revision object stores context data [Bailey et al. 2002]. It provides an indicator for the circumstances under which the object revision should be retrieved. Context data can take many forms. For example an object revision's context data may list the revision's creation date, or the age of the client for whom the object is suitable (e.g. in respect of child, teenager or adult). Which object revisions are selected is determined at runtime by matching the client's context (data) against the object revision's context data [Michaelides et al. 2001; Weal et al. 2001].

Figure 6.12 shows the context data enclosed within Context objects (represented by hexagon objects). Context data can have a direct relationship with versioning as the context data of an object revision can be used to store versioning information. Figure 6.12 shows an example where the context data is used by each revision to specify whether it is the first or second revision of Reference object R1.

6.4.2.3. Limiting Revision Proliferation

Contrary to the OHP-Version protocol approach, FOHM versioning removes object reference embedded information that points objects to specific objects revisions. This means that no existing objects need to be updated whenever a new revision of an existing object is appended to a FOHM structure. Thus no new revision of an existing object needs to be created since no existing objects are being modified. Consequently revision proliferation is prevented.

Figure 6.13 shows an example. Adding a new revision of Reference object R1 to the existing structure of Figure 6.12 leads to only one new revision being created shown as 'R1 v3'. No other objects need to be created.

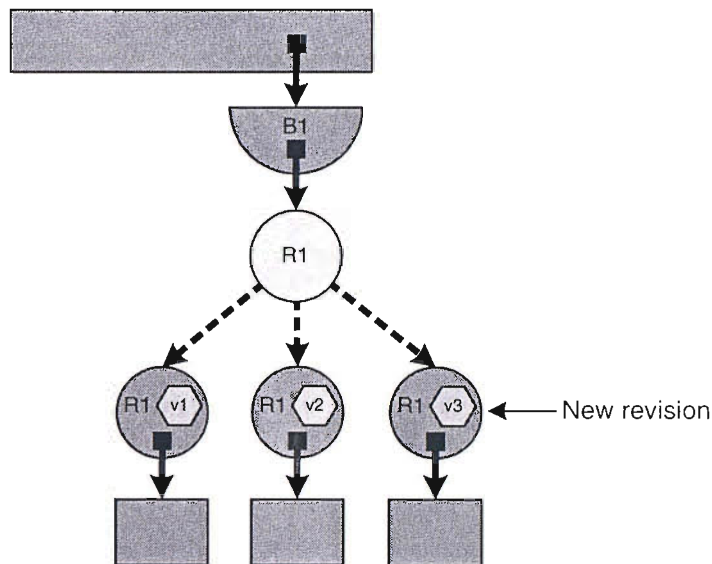


Figure 6.13: Adding a new revision to the existing FOHM structure of Figure 6.12 during the Contextual Connection process.

However the Contextualised Connections process does not eliminate revision proliferation completely as demonstrated in the next section.

6.4.2.4. Embedding Still Causes Revision Proliferation

Revision proliferation still occurs when attempting to append a new object to an existing FOHM hypermedia structure. This time it is when the new object is not a new revision of an existing revision set, but is a totally brand new object. The problem is that object references continue to be embedded as part of a FOHM object's content.

Revision proliferation results because any existing FOHM object that is to reference the new object must have its connection data (i.e. embedded object reference) updated in order to reference the new FOHM object. But this means that the existing FOHM object's content is being modified. Therefore versioning policy dictates that a new copy must be created of that updated object. Thus a new (and unnecessary) copy of that object will have to be created. The original copy preserves the original embedded connection data, whilst the new copy contains the new reference.

Figure 6.14 shows an example. If new Reference object R2 (which is not a revision object of existing Reference object R1, but an altogether new object) is to be appended to Binding object B1, then B1 has to be updated to include an embedded reference to Reference object R2.

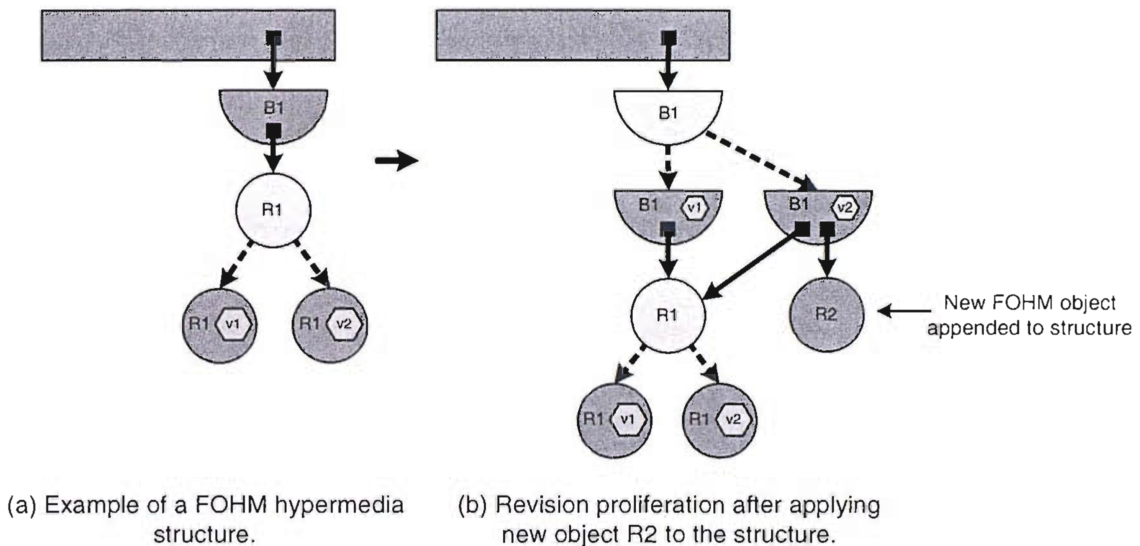


Figure 6.14: Contextualised Connections causing revision proliferation.

For those reasons as already stated in Section 6.4.1 the creation of new objects just to store changed object references is not good versioning practice. The creation of the new attached object revision is illogical. Virtually the same action as appending a new object revision to the structure (via Contextualised Connections) has been enacted, but the result is a different structural formation (compared to that generated by Contextual Connections).

6.5. Issue Four: Hypermedia Structure Maintenance

The problem of structure maintenance, as described in Chapter 5, is a general one. The Solent CB-OHS and Auld Linky are no exception to suffering the three structure maintenance problems described in Section 5.3: dangling hypermedia structures, specious hypermedia structures and misaligned internal references.

No research has (as yet) been undertaken into considering structure repair strategies for either OHP-Nav or FOHM once structure has become broken. But what is apparent is that, no matter what repair scheme may be investigated in the future, it

will certainly be adversely affected by the nature of OHP-Nav and FOHM's connections between objects. I.e. the embedding of connection information (between objects) within the hypermedia objects themselves. Such embedding impinges efficient and logical repair of broken structure.

In particular it can cause the repair of OHP-Nav or FOHM structures comprised of more than one internal route to be especially confusing. This was one of the main reasons as to why the OCS data model was developed.

6.5.1. Multiple Internal Routes Within Structure

An example of a hypermedia structure that comprises more than one internal route is shown in Figure 6.15. This diagram was first introduced in Section 5.3.4.

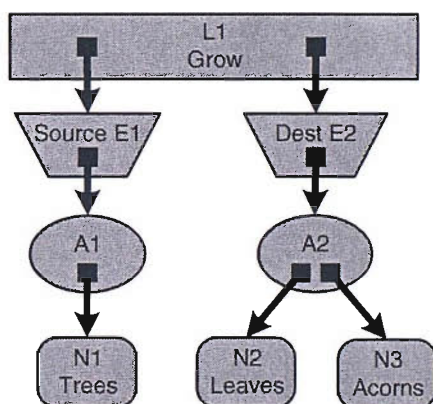


Figure 6.15: Hypertext link containing two internal routes.

The example OHP-Nav hypertext link of Figure 6.15 has two internal routes. The first route expresses 'Trees Grow Leaves'. It is composed of objects N1, A1, E1, L1, E2, A2 and N2. The second route shows 'Trees Grow Acorns'. It is composed of objects N1, A1, E1, L1, E2, A2 and N3.

6.5.2. Broken Internal Routes

A broken structure problem specific to internal routes is when one route breaks in such a manner that the direct repair of that broken route will have an adverse effect on any other unbroken routes contained within the same structure.

The example first introduced in Section 5.3.4 which involved Figure 6.15 will now be expanded upon. The example centres on the content of the 'Acorns' document having

been modified in such a way that Anchor A2 now points to incorrect content within the 'Acorns' document. Thus as far as the link to the 'Acorns' document is concerned Anchor A2 needs updating. However the 'Leaves' document has not been modified, therefore the link to the 'Leaves' document does not require Anchor A2 to be updated as A2 continues to point to the correct content. The upshot is that A2 cannot be directly repaired (for the benefit of the 'Acorns' document link) without negatively affecting the 'Leaves' document link. Consequently the internal route structure of the (overall) hypermedia structure must be re-arranged in order to both repair the structure and maintain the status quo. The desired repaired link solution is shown in Figure 6.16.

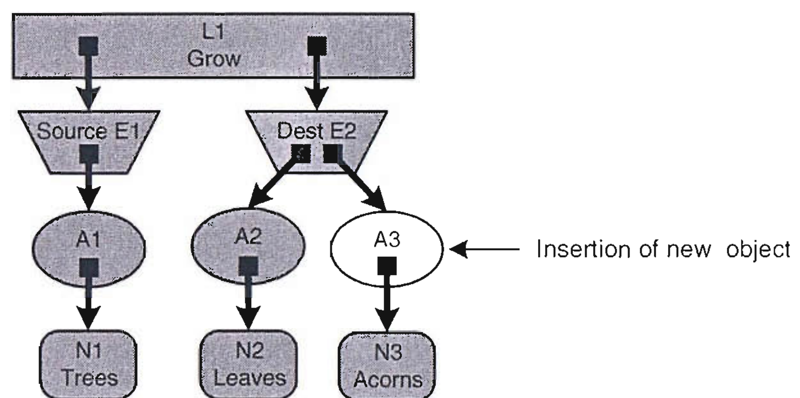


Figure 6.16: Repaired hypertext link.

The first thing to notice is that no functional content of any pre-existing hypermedia object needs amending. Instead, the update to Anchor A2 is resolved by creating a new copy of it, and then making the update to that new object. This new hypermedia object (shown as Anchor A3) is necessary so that the 'Acorns' document is referenced correctly. Original Anchor A2 must continue to exist so that the 'Leaves' document is referenced correctly. Finally, the overall arrangement of the hypertext link must be updated so that the two internal routes (within the hyperlink) connect the appropriate documents together. Anchor A2 is disconnected from Node N3, and new Anchor A3 is connected to Endpoint E2 and Node N3.

6.5.3. A Confusing Repair Process

The act of re-arranging a hypermedia structure (such as those of OHP-Nav or FOHM) which comprise multiple internal routes can be a potentially confusing process. This is because the hypermedia objects of these structures contain embedded object references and it is their update that can often throw up unwelcome surprises for the client. This is for two reasons.

1. The connection information is embedded in some object types, but not others. For example the connection data between OHP-Nav Anchor and Endpoint objects is stored in Endpoints, but not Anchors. Thus clients must be aware of which objects to modify when connecting hypermedia objects together when repairing hypertext links. This can cause confusion as it leads to some objects being updated whilst others are ignored.

2. Having to update the contents of objects purely to establish connections between objects can muddy the practice of link maintenance. This is because the client will often be required to make additional updates to the contents of objects that are not obvious judging by the structure maintenance solution. Therefore it can be puzzling as to why certain hypermedia objects are being updated.

The link maintenance solution of Figure 6.16 provides examples of both potential problems described above when OHP-Nav hypertext links are composed of objects containing embedded object references.

In order that the objects of the hypertext link can be re-arranged from the first formation of Figure 6.15 to the second formation of Figure 6.16 it is necessary to update Endpoint E2. This is not because any functional aspect of the object needs to be changed, e.g. the internal route's direction, but because Endpoints store the Endpoint-to-Anchor connection data. Hence existing Endpoint E2 needs to be updated to enable new Anchor A3 to be attached to it.

This clearly interferes with the pellucidity of the structure maintenance process as the client must have the knowledge that in order to connect Endpoint E2 to Anchor A3 that it is E2 that has to be updated with the relevant connection information and not Anchor A3. Moreover such object content modifications are confusing since the link maintenance solution makes no indication that the internal object content of Endpoint E2 should be modified. Therefore the need to and the act of updating this object may come as a complete surprise.

The client must also update the content of original Anchor A2. This is despite that the functional content of this object is not meant to be updated since a new object has been created in its place (object A3). But this modification is necessary so that A2 is no longer connected to Node N3. It has to be updated, because it is Anchors that store the connection data between Node and Anchor objects. Again the client must be aware that it is the Anchor and not the Node that must be updated. Once again this is another source of confusion because the whole point of the link solution is that

original Anchor A2 is left intact since it is needed for the 'Trees Grow Leaves' document route.

6.6. Summary

This chapter has listed the four issues that led to the creation of the Object and Connection Space (OCS) data model. They were found through combined experimentation with OHP-Nav structure and the arbitrary structure served by the Solent CB-OHS, and FOHM structure served by the Auld Linky CB-OHS.

The first issue under investigation is the inefficient utilisation of individual hypermedia objects. The conventional representation of hypermedia structure severely limits the re-use of individual hypermedia objects. This means that many hypermedia objects are used and stored that fulfil the same role which is an unnecessary waste of resources. The OCS data model solution to this problem is covered in Chapters 7 and 8.

The second issue discussed is the inability to re-use entire segments of structure repeated within other hypermedia structures. Once again this is an inefficient utilisation of resources especially as FOHM structures can be very large making the prospect of repeating hypermedia structures that much more likely. The OCS data model solution to this problem is also covered in Chapters 7 and 8.

The third issue is the problem of revision proliferation. Namely the needless creation of new revisions of hypermedia objects when carrying out normal hypermedia object versioning. OHP experimentation in the form of the OHP-Version protocol highlighted the full extent of this problem. The FOHM solution, in the form of Contextualised Connections, limits revision proliferation somewhat, but did not completely eradicate it. Thus the problem of revision proliferation persists. The OCS data model answer is described in Chapters 9 and 10.

The final issue is the topic of structure maintenance. Embedding data object references in hypermedia objects obscures clarity when repairing broken hypermedia structure in particular when repairing broken internal routes. How the OCS data model tackles this problem is explained in Chapter 11.

Chapter 7. Opening Hypermedia Structure

7.1. Introduction

This chapter describes the basic concepts behind the Object and Connection Space (OCS) data model. Also examined are the OCS data model's aims and objectives, namely to open up hypermedia structure to enable the re-use of hypermedia objects and the connections between objects. This is in an effort to generate hypermedia structures composed of re-usable objects. The net result is a prevention of wasteful and unnecessary duplication of resources. A key benefit is that it also lessens the burden of general hypermedia object storage.

In the process this chapter identifies how the OCS data model addresses Computer Science Contributions 1, 2 and 3 (of Chapter 1) and Problem Domain Issues 1 and 2 (of Chapter 6).

CSC1:	Extending the concept of open hypermedia into the realm of hypermedia structure.

CSC2:	Promoting the general re-use of hypermedia structure.

CSC3:	Offering a more logical approach to hypermedia structure representation.

PD1:	The use and storage of hypermedia objects that carry out the same function.

PD2:	The use and storage of identical hypermedia structure.

7.2. Unopen Hypermedia Structure

Hypermedia structure has a dual role. One role is to provide functionality (the functional role), and a second role is to connect items together (the connectional role). Items for connection can be documents, or they can be the individual hypermedia objects that make up hypermedia structure. It is the latter role (the connecting together of hypermedia objects) that is of interest as regards the OCS data model.

- *Functional role.* Expresses the function of an object. For example an OHP-Nav Anchor object describes the content (within a node resource object) referenced by an OHP-Nav hypertext link.

- *Connectional role.* Describes which other objects a given hypermedia object is connected to. For example to which OHP-Nav Endpoints and/or OHP-Nav Node objects a given OHP-Nav Anchor object is connected.

The typical approach adopted by conventional hypermedia objects, e.g. OHP-Nav and FOHM objects (Sections 3.2.3 and 3.3), is to combine both the functional and connectional roles within each hypermedia object. This is through the embedding of connection data (i.e. object references) within the same hypermedia objects that are also used to record the function of the object. Such object reference embedding is described in Sections 3.2.5 and 3.3.4.

However there is no particular reason why such embedding is the norm. It is usually an oversight by hypermedia structure data model designers. Their focus is so often concentrated on getting the high-level structural abstractions to work, e.g. hypertext links, that attention is rarely given to the lower level aspects of the structure, i.e. the connections between hypermedia objects. At least that was the case until the development of the OCS data model.

Nonetheless the problem with combining the functional and connectional roles within single objects is that it leads to *closed* hypermedia structure. Clearly this stands in direct contrast to the aim of *openness* within an open hypermedia environment (Section 2.12).

Conventional hypermedia structure can be regarded as closed according to this definition of what makes a hypermedia system open:

An Open Hypermedia System is open, because it is able to link to and from document content without altering the document content itself.

By the same token:

Hypermedia structure can be said to be open if it can link to and from other structure without having to alter the (linked) structure itself.

As already stated, a major role for hypermedia structure is to connect hypermedia objects together. Thus hypermedia structure is already capable of linking to and from structure. But when linking to existing structure, that existing structure must also be altered. Hence conventional hypermedia structure is considered closed. This is because in linking to existing structure, that structure's connectional data (describing to which objects it is connected) must be updated.

The benefit to be gained if structure is open is the opportunity to re-use existing hypermedia structure. This not only makes for efficient use of structure, but it also benefits hypermedia versioning (Chapter 9) and hypermedia structure maintenance (Chapter 11). Moreover re-use is a crucial feature of open hypermedia (Section 2.12), therefore it is only natural that hypermedia structure should also be re-usable too. Chapter 8 in particular describes the advantages when opening up the structure of OHP-Nav and FOHM hypermedia objects.

7.3. OCS Data Model Objectives

The aim of the OCS data model is to address the internal organisation of hypermedia structure. Specifically on how hypermedia objects are connected together. To this end the OCS data model separates the object and connection data as recorded within hypermedia structure. Separation enables re-use of the same hypermedia object and connection data to construct re-usable hyperstructure. That hypermedia structure can then be used to form new relationships between different node resource objects within hypermedia networks. The immediate benefits are saving on unnecessary duplication of resources. The same hypermedia objects and connections (between hypermedia objects) can be shared by multiple (possibly unrelated) hypermedia structures. It is through separating hypermedia objects and connections that enables the Computer Science Contributions of Chapter 1 and the Problem Domain Issues of Chapter 6 to be realised.

7.4. C-Level Work

Looking at the OCS data model from the point of view of Engelbart's A-B-C level of work [Engelbart 1998; Nürnberg 2002], the research on the OCS data model can be categorised as C-level work. This is where A-level is the primary work, B-level supports the primary work, and C-level provides support for the support work. On its own the OCS data model does not fulfil the primary work of an organisation. Its purpose is to act as a support tool to enable the primary work to be carried out. That is, the OCS data model is a tool of the hypermedia system for ensuring that the internal hypermedia objects of hypermedia structure are organised to their optimum so as to improve the hypermedia system's overall efficiency.

Through its low-level activities the OCS data model aims to benefit structure manipulation at all layers of the Open Hypermedia System architecture: Client Application Layer, Middleware Layer and Storage Back End Layer (Section 7.15).

7.5. Drawing on OHP-Nav

The thesis mainly looks to the OHP-Nav data model (Section 3.2) for example hypermedia structures upon which the OCS approach can be applied. However, the Object and Connection Space is not restricted to just this data model as explained in Chapter 8. The OHP-Nav data model was chosen as it is a hypermedia structure representation with which I am very familiar as I have used it to develop Open Hypermedia Systems [Reich et al. 1999b; Millard 2000a] and research hypermedia concepts [Griffiths et al. 1999; Griffiths et al. 2002]. Moreover the OHP-Nav data model is a recognised data model within the hypermedia community [Nürnberg 1999].

7.6. "Rock, Paper, Scissors"

For the most part the examples used throughout the rest of the thesis are based on the "Rock, Paper, Scissors" game (abbreviated to the RPS game) [World RPS Society 2002]. The game consists of two players who challenge each other by shaping their hands to form one of either rock, paper or scissors. The contest has six possible outcomes shown by Figure 7.1.

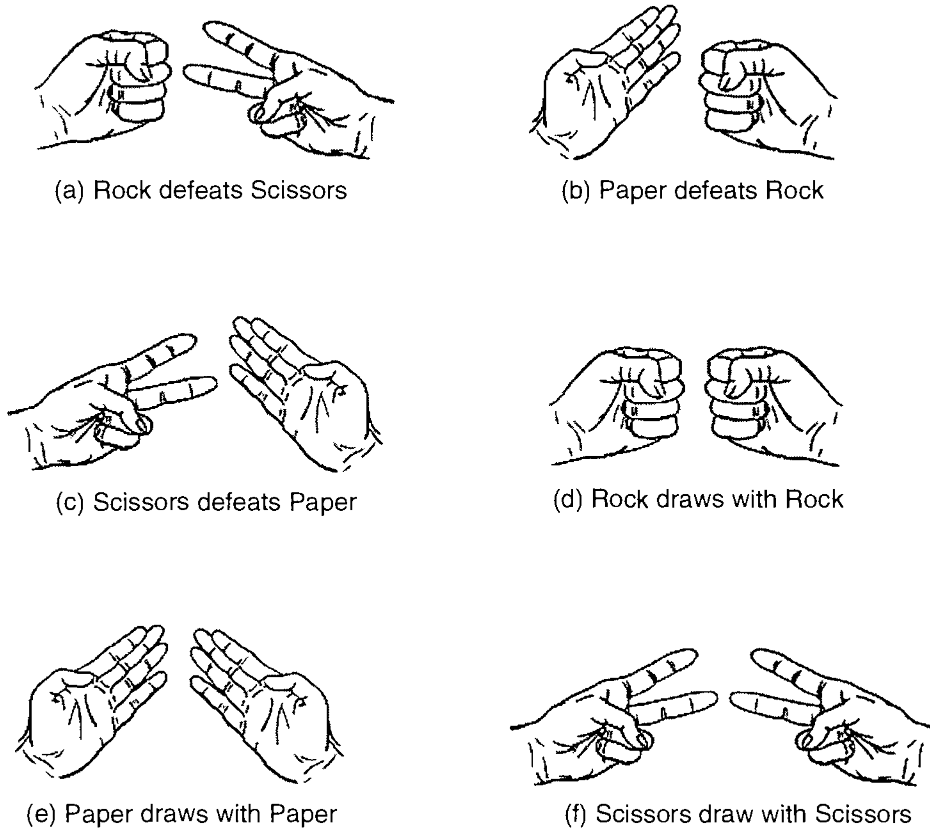


Figure 7.1: 'Rock, Paper, Scissors' outcomes.

Each outcome of the RPS game can be modelled as a different hypermedia structure (Figure 7.2). Such hyperstructures provide a useful demonstration for the benefits of the Object and Connection Space approach when generating re-usable hyperstructures.

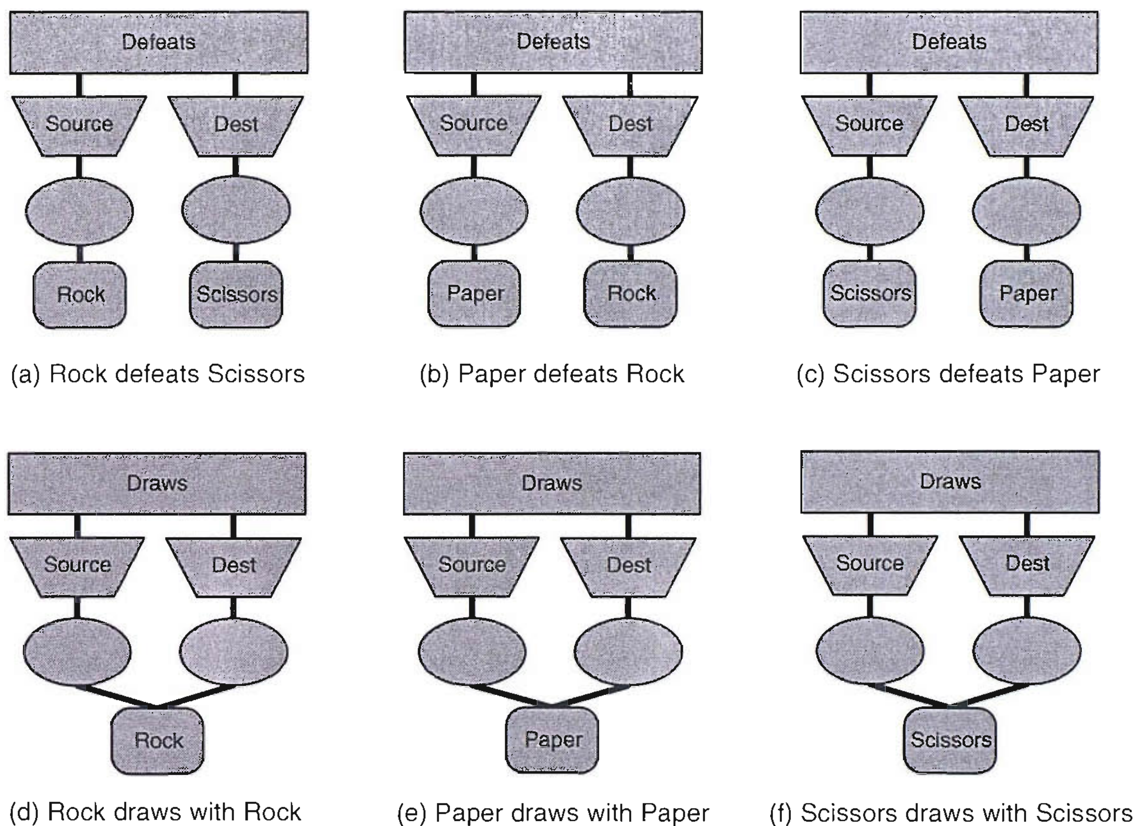


Figure 7.2: 'Rock, Paper, Scissors' hyperstructure outcomes.

Looking Figure 7.2 it is evident that re-use opportunities already exist within the six "Rock, Paper, Scissors" hypermedia structures. For example functionally all the 'Defeats' Link objects are the same, all the 'Source' Endpoint objects are the same, all the 'Destination' Endpoints are the same, all the unmarked Anchors are the same, all the 'Rock' Nodes are the same, all the 'Paper' Nodes are the same and all the 'Scissors' Nodes are the same.

An example of their direct re-use is shown in Figure 7.3 where all hypermedia objects that can be re-used have been re-used. As can be seen such an arrangement does not make it clear as to how the objects are being re-used in respect of the relationships they are trying to show, e.g. 'Rock Defeats Scissors' or 'Rock Draws with Rock'. Moreover it also creates incorrect relationships, such as 'Scissors Defeats Rock'. This was one of the reasons why the OCS data model was devised so that re-used hypermedia objects can be organised in such a fashion so that there can be no doubt as to what structural relationships are being expressed within a given hypermedia structure.

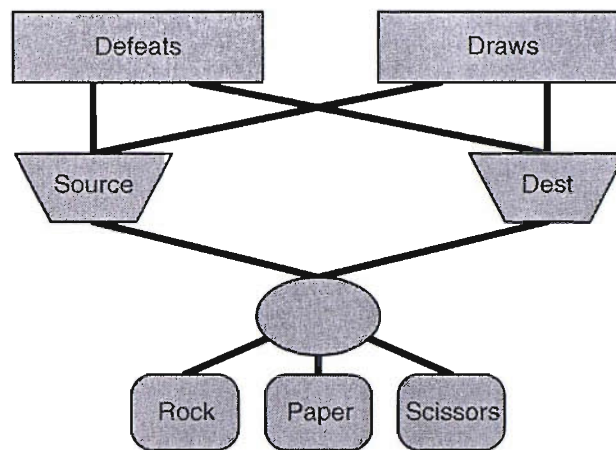


Figure 7.3: All RPS hypermedia objects are subject to re-use, but they fail to make clear the relationships they are trying to express.

In respect of the hypermedia structure examples used throughout the thesis, the thesis focuses on just the three 'Defeats' relationships. This is because they provide the most interesting cases for hyperstructure re-use. The three 'Draws' relationships have only been included in Figures 7.1, 7.2 and 7.3 for completeness. Re-use as regards their hyperstructure formations is not considered hereafter.

7.7. The Object and Connection Space Data Model

The OCS data model opens up hypermedia structure (Computer Science Contribution 1) by allocating the hypermedia objects of a hypermedia structure to a Function Object Space (Section 7.7.1) and the connections between hypermedia objects to a Connection Space (Section 7.7.2). In line with the premise of open hypermedia [Reich et al. 1999a] and Structural Computing [Reich et al. 1999a; Nürnberg and Schraefel 2003], hypermedia objects are promoted as first class entities. They have their own identity and are referencable.

The connections between hypermedia objects are also promoted as first class entities. They have their own identity and are referencable. It is this allocation to Object and Connection Spaces that opens up and hence facilitates the re-use of hypermedia objects and connections.

In many respects the OCS approach is applying the principles of open hypermedia separation to the internal organisation of hypermedia structure, i.e. the separation of linking information from being embedded within node data. This idea is expanded upon in Section 7.7.3.

Figure 7.4 shows the example hyperstructure used throughout the rest of this chapter to demonstrate the capabilities of the OCS approach. It depicts the relationship 'Rock defeats Scissors'.

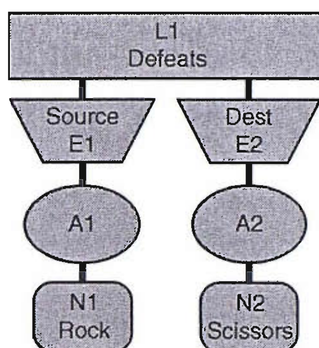


Figure 7.4: Hypermedia structure 'Rock defeats Scissors'.

7.7.1. The Function Object Space

The Function Object Space records the hypermedia objects that are used within one or more physically created hypermedia structures. Figure 7.5 shows an example of a Function Object Space. It records all the objects that make up the OHP-Nav hyperstructure of Figure 7.4.

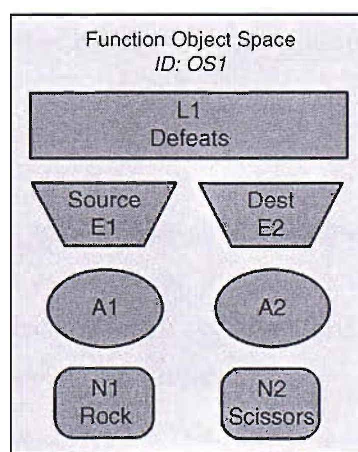


Figure 7.5: Example of a Function Object Space.

Once assigned to the Function Object Space, objects can be subjected to re-use by Connection Spaces (Section 7.7.2). Hypermedia data objects after they have been assigned to the OCS are referred to as Function Objects. This is because these object

types express the functional role of a hypermedia object, e.g. the functional role of an OHP-Nav Endpoint object is to describe the overall direction of a hypertext link. Thus OCS Function Objects are virtually the same as conventional hypermedia objects. The main difference being that they do not record any connection information. Function Objects, like conventional hypermedia objects, continue to serve as first class entities with their own identity.

Appendix A lists an XML specification for OCS Function Objects (Section A.3). The appendix also includes examples of Function Object instances in Sections A.5 and A.6.

7.7.2. The Connection Space

The Connection Space records specific connections between Function Objects. They are captured within a special type of object called a Connection Object. They assume the connectional role of a hypermedia object (see Section 7.2). Connection Objects are lightweight objects compared with Function Objects since they only contain connection information. In this regard Function Objects are considered heavyweight objects (this is explained in Section 7.9).

Appendix A specifies the XML specification for OCS Connection Objects (Section A.4) along with examples of actual Connection Object instances (Sections A.5 and A.6).

Connection Objects can consist of binary or n-ary connection formations.

7.7.2.1. Binary Connection Objects

A binary Connection Object contains a single connection between two Function Objects. Figure 7.6 shows an example of the connections of the hyperstructure of Figure 7.4 broken down into binary connections. (Section 7.8 explains the diagrammatic depiction of Connection Objects.)

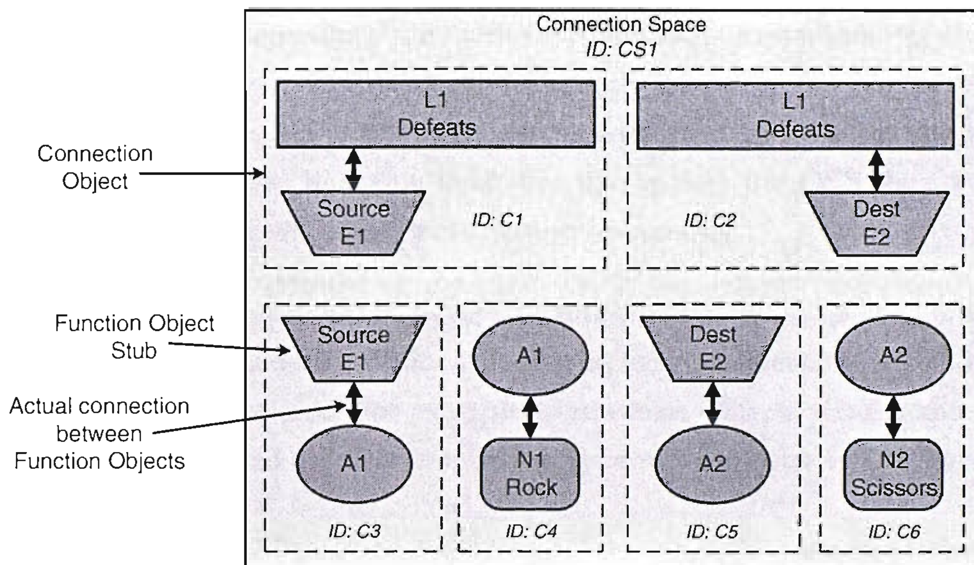


Figure 7.6: Connection Space composed of binary Connection Objects only.

7.7.2.2. N-ary Connection Objects

An n-ary Connection Object contains several connections between Function Objects. Figure 7.7 shows an example of the connections of the hyperstructure of Figure 7.4 broken down into a variety of binary and n-ary connections.

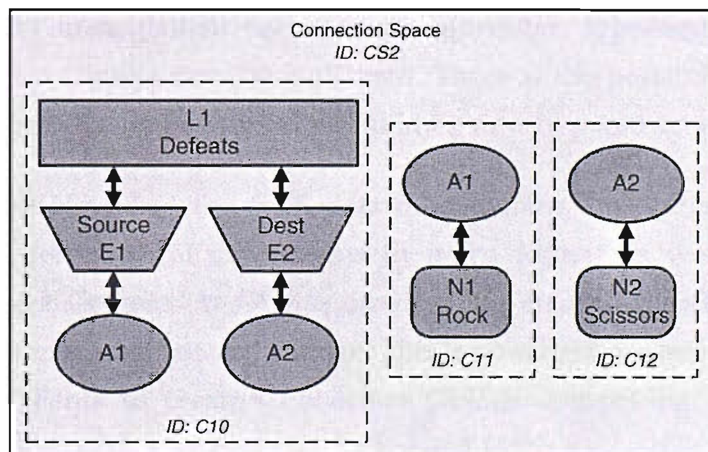


Figure 7.7: Connection Space composed of binary and n-ary Connection Objects.

7.7.2.3. Which to Choose: Binary or N-ary Connections?

The capability of representing a hyperstructure's connections as both binary and/or n-ary formations is necessary as each connection formation offers both advantages

and disadvantages. Depending on which connection formations are chosen influences the type of re-usability that can take place. It is left to the discretion of the client to determine which hyperstructure connection formation is most appropriate for their re-use purposes. It is this flexibility that makes the OCS data model a powerful mechanism for efficient hyperstructure re-usability.

The ideal solution is to represent all hyperstructure connections as binary Connection Objects. This offers the most flexibility as it enables every connection (within a given hyperstructure) to be re-used. For example Connection Objects C1, C2 and C3 (of Figure 7.6) could be joined together to form the hyperstructure shown in Figure 7.8.

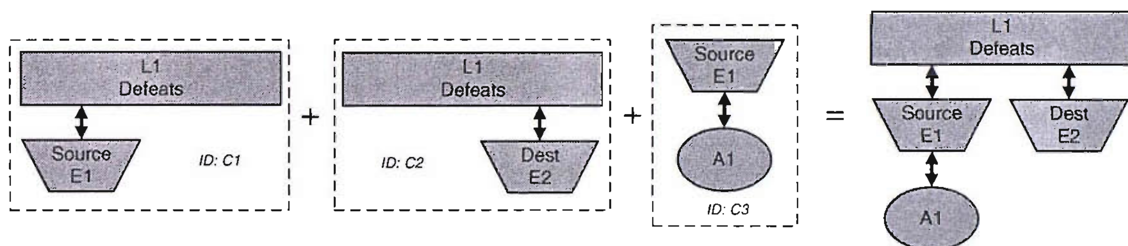


Figure 7.8: Example of connections being re-used.

The disadvantage with representing all hyperstructure connections in binary format is that many Connection Objects will often have to be created (one Connection Object for every connection within the hyperstructure). This can substantially increase the amount of object management and storage. Moreover, representing structure as binary Connection Objects may be inefficient. There is the possibility that many of the connections may never be re-used for building new hyperstructure formations.

It is for these reasons that the OCS data model offers the variable approach of representing hyperstructural connections in n-ary format as well. This solution means that fewer Connection Objects need to be created thereby reducing the amount of object management and storage. This is possible because n-ary Connection Objects enable clients to create Connection Objects comprising just the specific connection arrangements guaranteed to be re-used in the future. For example a client may only want to re-use the structural arrangement represented by connection object C10 of Figure 7.7 hence it is stored as a single entity.

However the disadvantage with n-ary Connection Objects is that they have the potential to limit hyperstructure re-use as it may not always be possible to re-use the individual connections between Function Objects. For example the individual connection between Function Objects L1 and E1 (of Connection Object C10 of Figure

7.7) cannot be re-used as a connection in its own right as it is now a member of the larger Connection Object C10. If that Connection Object is to be re-used, then all the Function Objects of the Connection Object (L1, E1, E2, A1 and A2) will also be re-used as well.

To solve this problem all the client has to do is create new Connection Objects that describe the desired hyperstructural connections. Even though more Connection Objects have been added to the Connection Space this is not a particularly "object expensive" solution as Connection Objects are deemed lightweight objects as explained in Section 7.9.

7.7.3. Comparison with OHS Hypertext Links

As stated at the beginning of Section 7.7 the OCS approach can be regarded as applying the open hypermedia principle of separating links from nodes to the internal organisation of hypermedia structure. Figure 7.9 shows how the individual objects and connections between objects (within hypermedia structure) may be viewed as being equivalent to separated OHS nodes and links.

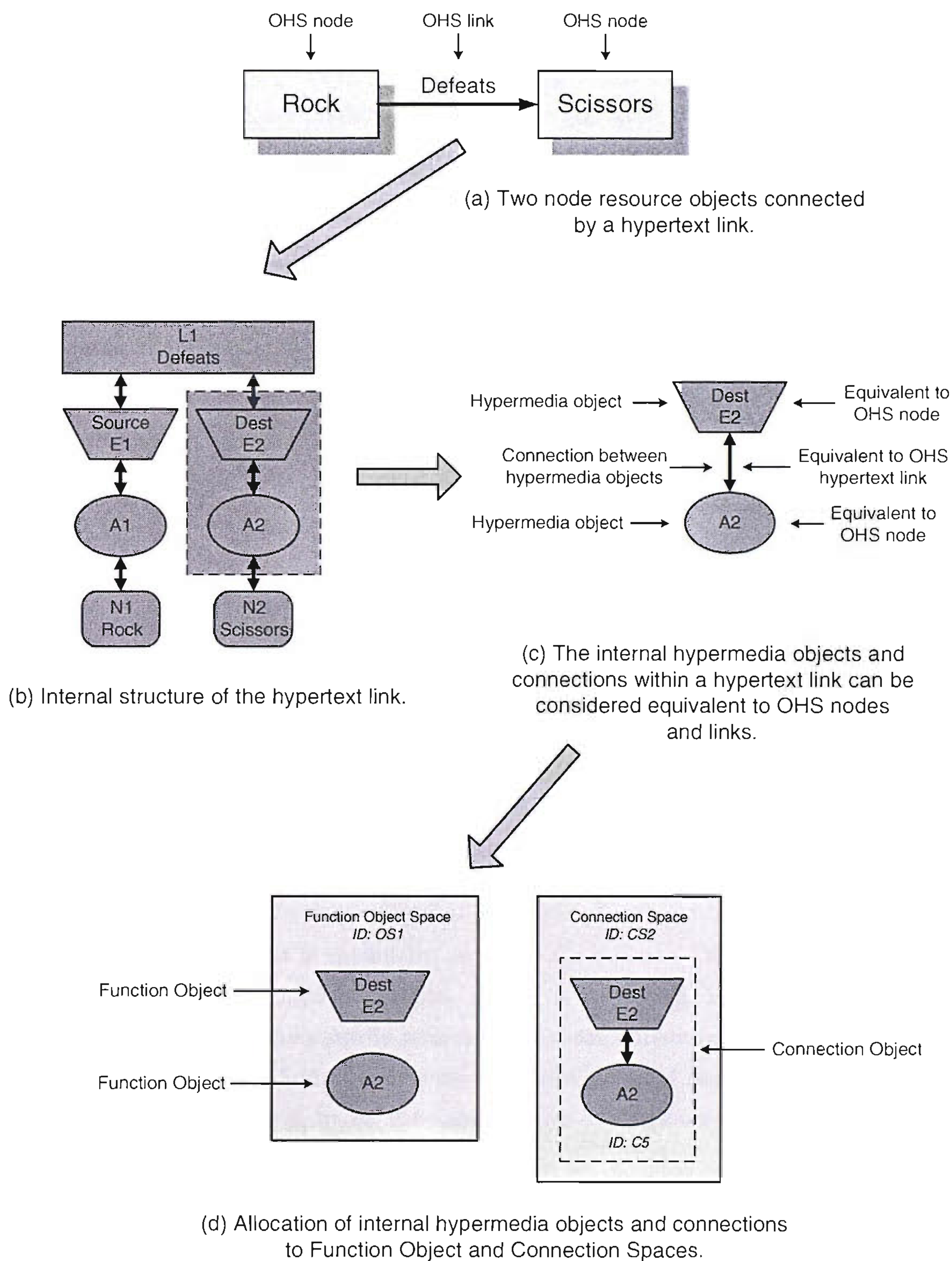


Figure 7.9: Comparison of OHS node and links with OCS Function and Connection Objects.

An obvious comparison is between Function Objects and OHS nodes. This is because, like OHS nodes, Function Objects do not contain embedded linking data that state to which other Function Objects it is connected.

Another obvious comparison is between binary Connection Objects (Section 7.7.2.1) and OHS hypertext links. Like OHS links, binary Connection Objects are separated from the resources (i.e. Function Objects) they are used to connect, and like the majority of OHS links, Connection Objects are used to form connections between two resources only.

However, n-ary Connection Objects (Section 7.7.2.2) are somewhat different to conventional separated OHS hypertext links. When an OHS hypertext link is considered n-ary, it usually means that it is a single link emanating from a single node resource pointing to two or more node resources (e.g. Figure 7.10).

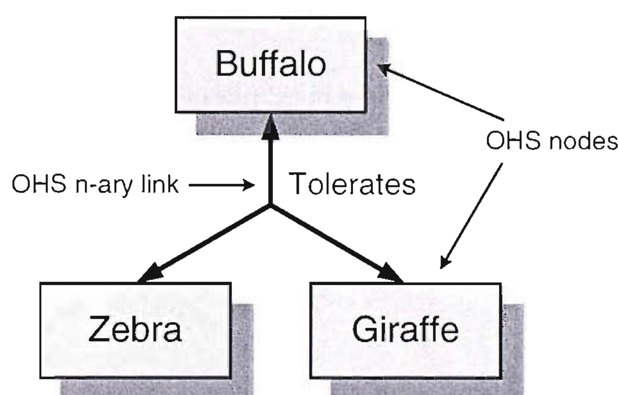
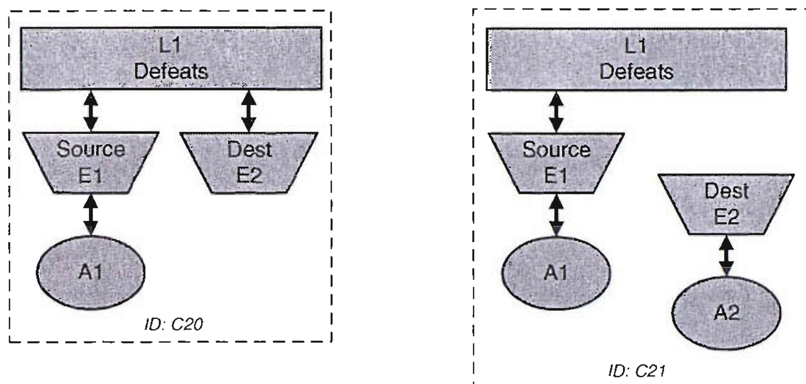


Figure 7.10: An example of an n-ary OHS hypertext link.

This is not usually what is meant by an n-ary Connection Object arrangement. A Connection Object is typically taken to be a grouping together of multiple connections that do not necessarily reference the same Function Object. Normally the connections link all the Function Objects into some form of sequence, although this does not necessarily have to be the case. Figure 7.11 shows an example of both scenarios.

In this respect an n-ary Connection Object is less like a hypertext link and more like a hypertext network. A simple example of a hypertext network is shown in Figure 7.12 for comparison.



(a) N-ary Connection Object that connects all Function Object stubs within the Connection Object.

(b) N-ary Connection Object that does not connect all Function Object stubs present within the Connection Object.

Figure 7.11: Two examples of n-ary Connection Objects.

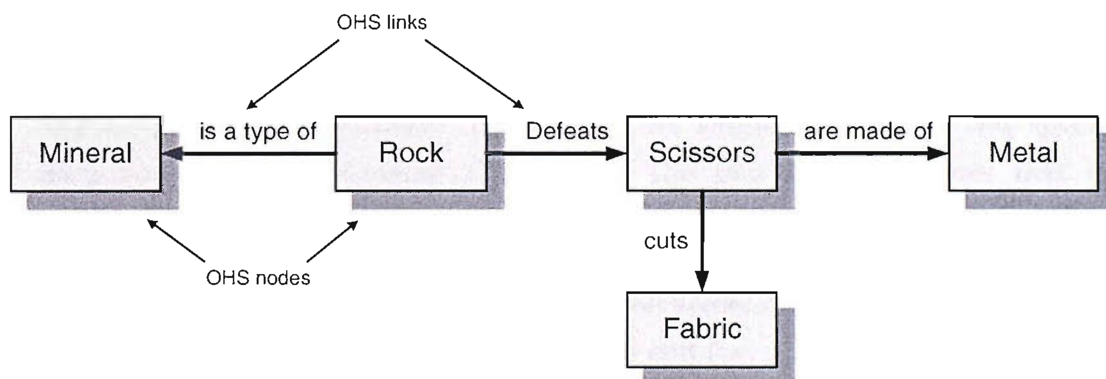


Figure 7.12: Example of a hypertext network.

7.8. Connection Objects Explained

There are a couple of points worth making as regards the diagrammatic depiction of Connection Objects.

- A Connection Object is denoted by a dashed box.
- Actual connections (within Connection Objects) are marked with an arrowhead at each end. The arrowheads signify the bi-directionality of Connection Object

connections. This means that the OCS data model permits traversal to and from any object member (of a Connection Object) in any direction.

- A Connection Object can be used to connect Function or Connection Objects together.

- Any Function or Connection Objects recorded within a Connection Object are object stubs. Hence all object members are referential. This is akin to Whitehead's Versioned-Object approach [Whitehead 2001b] where the relationship between version history containers (read Connection Object) and members (read internal Function and Connection Object members) are referential.

- The OCS data model makes no assumptions about how connections are formed between object stubs. Referencing individual object stubs can be by specific object IDs or by some form of attribute data. An example of the latter approach is an object stub connecting to other OHP-Nav Endpoint object stubs based on a particular navigational direction attribute that they might hold. The OCS data model also has no preference as to whether the connections between objects are hand-made or computed[7] [Ashman et al. 1997; Ashman 2000b]. This thesis adopts the simplest option whereby object stubs are connected together via specific Function and/or Connection Object IDs. The thesis also assumes that the connections are hand-made.

- Each connection to a Function Object always connects to an entire Function Object. This is indicated by each connection end (i.e. arrowhead) pointing at the whole Function Object and not inside the Function Object. The reason for this is because Function Objects are treated as opaque atomic entities.

- Each connection to a Connection Object can either point at (i.e. connect to) the entire Connection Object or it can point at (i.e. connect to) internal objects within the Connection Object. This latter capability is explained in Section 7.11.1.

---

[7] Hand-made connections are individual connections that are explicitly created by a user, whilst computed connections are those automatically created based on some sort of computation perhaps making use of attribute data.

7.9. Lightweight vs. Heavyweight Objects

Connection Objects are considered lightweight objects. That is they are assumed to be small in size. The only type of data they contain is connection information which generally does not amount to a great deal of data.

In contrast Function Objects are considered heavyweight objects as a Function Object can drastically vary in size, and in many cases be significantly larger than typical Connection Objects. Examples of actual Function and Connection Objects expressed in XML can be found in Sections A.5 and A.6.

Function Objects represent the functionality of hypermedia structures. And as such their content is determined by the hypermedia data model (e.g. OHP-Nav or FOHM) that represents the hypermedia structure of the hypermedia server. As the OCS data model can be employed to organise the structure of *any* hypermedia structure data model, then the potential size of Function Objects is not known in advance. Thus these objects may contain any content which can be any size.

For example it is possible for OHP-Nav Node objects to contain the actual document content that the hypermedia structure is being used to connect. Thus they can be very large indeed. Conversely they can be very small as they may only contain the URL reference pointing to the document content location. The important point is that a Function Object's size is a completely unknown factor compared with a Connection Object.

The OCS data model takes the deliberate stance that Function Objects will in most cases be larger than Connection Objects. Therefore, via its re-use capabilities, the OCS data model prevents the creation of Function Objects that duplicate the same hypermedia function. Thus fewer Function Objects need to be stored. This saves Back End storage space especially if they are large Function Objects. Also, smaller message sizes can be sent between OHS components where only one Function Object needs to be sent instead of many if that particular Function Object is re-used multiple times within the same message. This can be a significant saving of transmission time if that Function Object contains a lot of data.

7.10. Connecting Connection Objects

Critical to making hypermedia structure re-usable is the ability to connect Connection Objects together. Two methods are employed: *conjoinment* and *attachment* operations.

7.10.1. Conjoinment Operation

The *conjoinment operation* is a form of pattern matching [Michaelides et al. 2001; Griffiths et al. 2002]. It enables existing connections within Connection Objects to be joined to one another. The operation works by connecting two or more Connection Objects together via a common Function Object. Figure 7.13 shows the two Connection Objects with IDs C10 and C11 (of Figure 7.7) being conjoined via Function Object A1. This produces the new hypermedia structure shown as Connection Object C18.

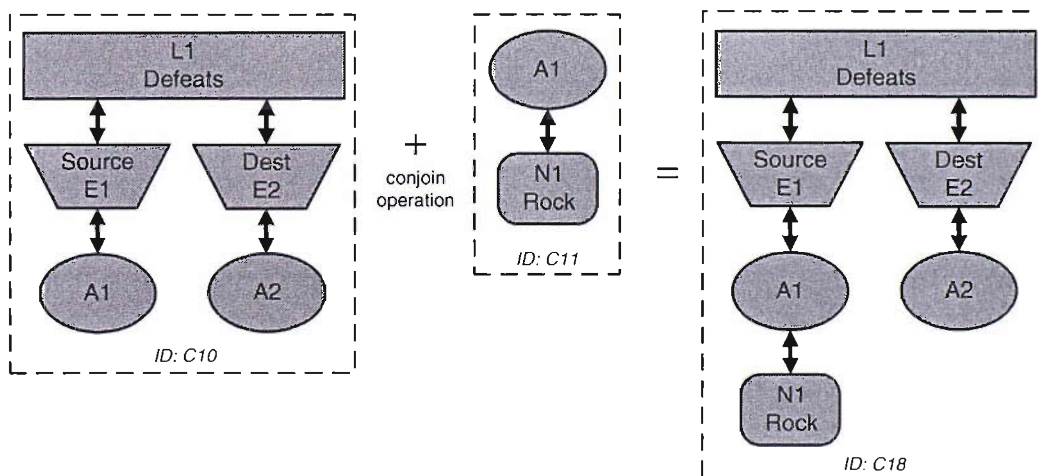


Figure 7.13: Conjoin operation.

7.10.2. Attachment Operation

The *attachment operation* creates new connections between Connection Objects and/or Function Objects. Figure 7.14 shows how two Connection Objects (of Figure 7.6) can be attached to create a new hypermedia structure. The Connection Objects are attached at Function Object E1 (of Connection Object C1) and Function Object A1 (of Connection Object C3).

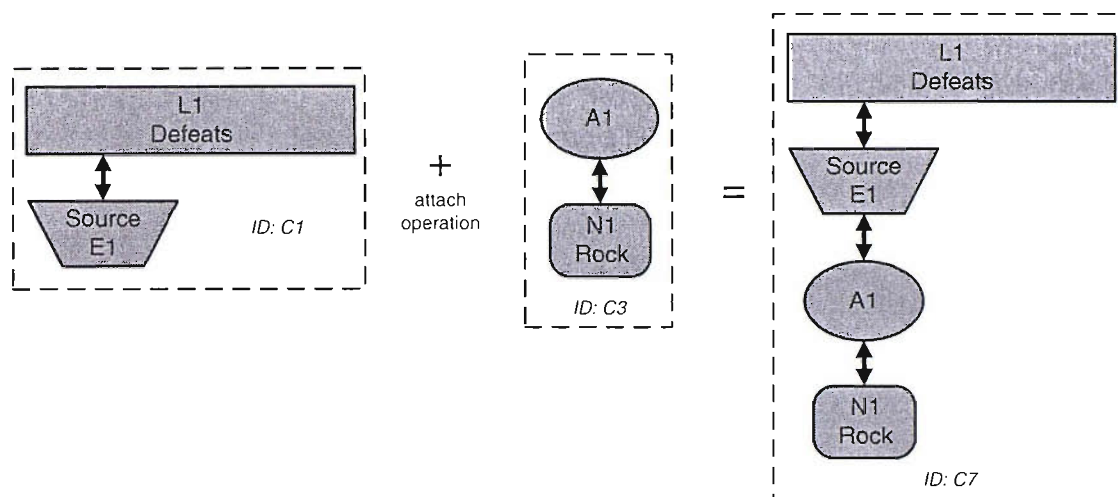


Figure 7.14: Attachment operation.

7.10.3. Linking To Hypermedia Structure

Both Connection Object operations (shown by Figures 7.13 and 7.14) demonstrate that the OCS data model fulfils Computer Science Contribution 1 – the opening up of hypermedia structure. This means that the OCS data model makes it possible for hypermedia structures to link to and from other hypermedia structures without having to alter either of the linked structures. This is because in each case neither of the existing structures (being conjoined or attached) need to be amended in order to produce the new structural formations. Instead any new arrangements are recorded within new Connection Objects.

7.11. Issues

The OCS data model brought forth a number of issues that need addressing. Two of the most pressing are described in this section.

7.11.1. Anchoring

The first issue is the problem of anchoring within Connection Objects. Some form of anchoring mechanism is necessary in order to join Connection Objects together. This is to enable Connection Objects to reference other Connection Objects and to reference Function Objects within Connection Objects.

Figure 7.15 provides an example scenario where a user wants to attach OHP Node N1 to OHP Anchor A1 of Connection Object C30. The question is how should OHP Node N1 reference A1 whilst it is a member of C30?

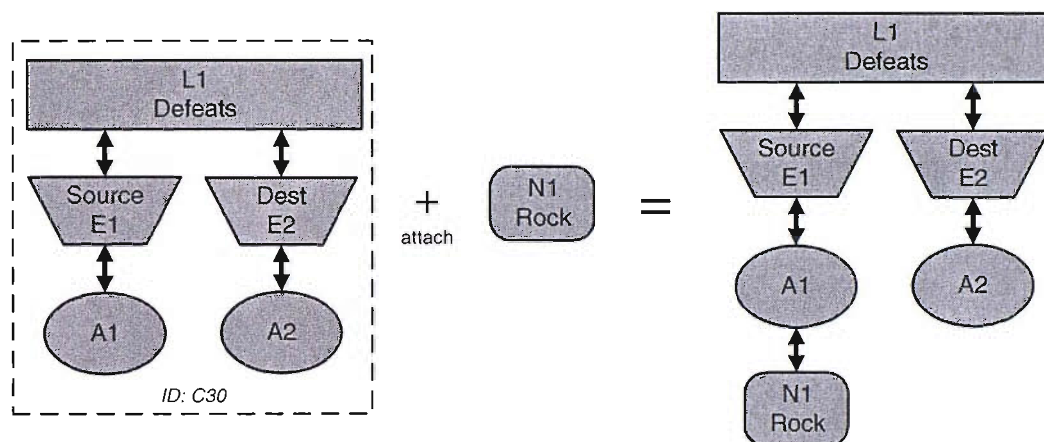


Figure 7.15: How should a Function Object be attached to another Function Object inside a Connection Object?

This is a similar problem to the anchoring problems faced by hypermedia data models in the past [Halasz and Schwarz 1994]. In these previous cases the question has been how to identify referencable regions within node object resources. Many different anchoring schemes have been devised. For example the OHP-Nav data model uses LocSpecs (location specifiers) [Reich and Millard 1999; Reich et al. 1999a; Reich et al. 2000]. They can choose anchors by using forward and/or backward axes to count the number of characters within text documents to a given location.

Likewise, many different anchoring solutions can be adopted within the OCS data model. For example Function Objects can be identified using just Function Object IDs. Or the position of a Function Object can be traced via tree-like navigation from the root Function Object. Alternatively, pattern matching might be used where the client describes the structural arrangement of the Function Objects against which to match.

The anchor solution adopted thus far takes the form of the traditional node and anchor referencing mechanism. The node argument specifies the Connection Object, and the anchor argument specifies either the Connection Object within a Connection Object or the Function Object itself. Function and Connection Objects are identified via their respective identifiers.

For example in order to locate Function Object A1 of Connection Object C30 of Figure 7.15, then the client would specify *{node: C30, anchor: A1}*. Figure 7.16 shows the attachment operation depicted by Figure 7.15. The heptagon object graphically represents a Connection Object. It is identified as Connection Object C30 by its node entry (shown as '*N*'), and the anchor by which other objects can attach (e.g. Node N1) is identified as A1 by its anchor entry (marked '*A*').

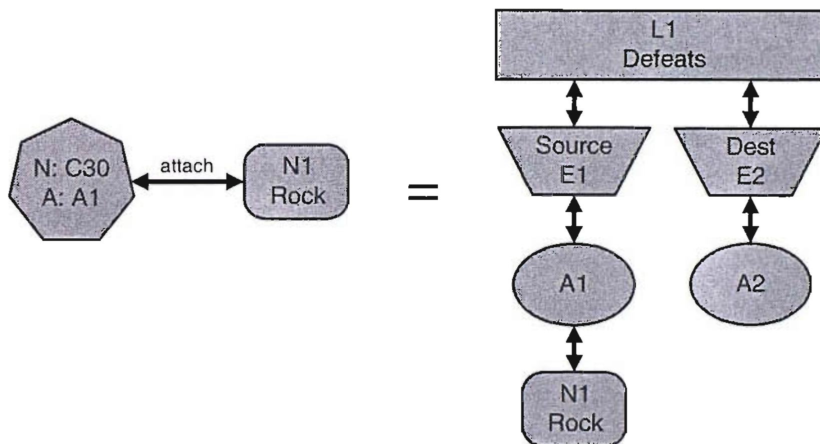


Figure 7.16: How a Function Object is attached to another Function Object inside a Connection Object.

7.11.2. Object Uniqueness

The ability to re-use Function Objects makes for efficient utilisation of hypermedia structure. However, it also presents potential difficulties when attempting to uniquely identify Function Objects which are being re-used. This problem manifests itself in two key areas.

7.11.2.1. Re-used Function Objects within a Single Connection Object

The first problem is when organising connections between Function Objects within a single Connection Object. The difficulty lies with how to identify which specific Function Object an internal connection is meant to reference when that Function Object is being re-used in more than one place within a Connection Object. An example demonstrates.

The idea is to create the hypermedia structure of Figure 7.17 where Function Object A1 is independently re-used in two places. In one place it is attached to E1 and N1,

and in another it is attached to E2 and N2. But because A1 is re-used by independent objects there is a danger of creating the incorrect hyperstructure of Figure 7.18.

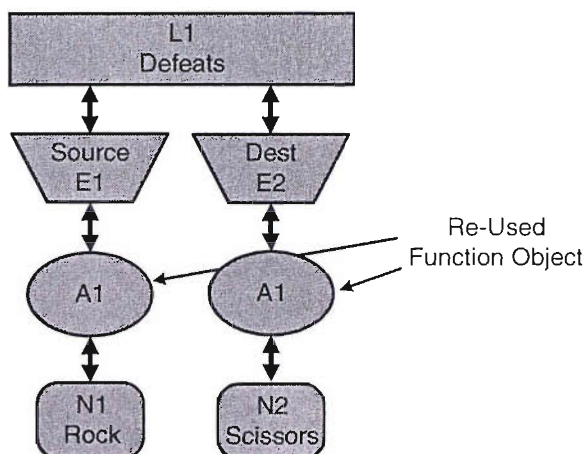


Figure 7.17: Desirable hyperstructure.

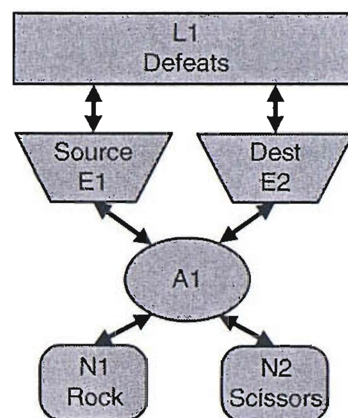


Figure 7.18: Undesirable hyperstructure.

The solution adopted by the OCS data model is the use of Instance Objects. An Instance Object is not a real object (like a Function or Connection Object), but a virtual Function Object. It works by a Function Object (that is being re-used) being referenced by a second identifier (called an Instance Object identifier) for every place (within a Connection Object) that the Function Object is being re-used. This allows clients to uniquely identify a re-used Function Object at the different places it is re-used within a given hypermedia structure.

Figure 7.19 shows the assignment of Instance Object identifiers to enable the re-use of Function Object A1. As A1 is used twice, there are only six Function Objects (in the Function Object Space) rather than seven which would be the usual case without re-use. Despite A1's re-use, its properties are not modified with any Instance Object identifier information. Instead this information is written to the Connection Space as it is to the Function Object stubs within Connection Object C30 that the Instance Object identifiers are assigned. This is shown by '*i1*' and '*i2*' markings being appended to both uses of Function Object A1.

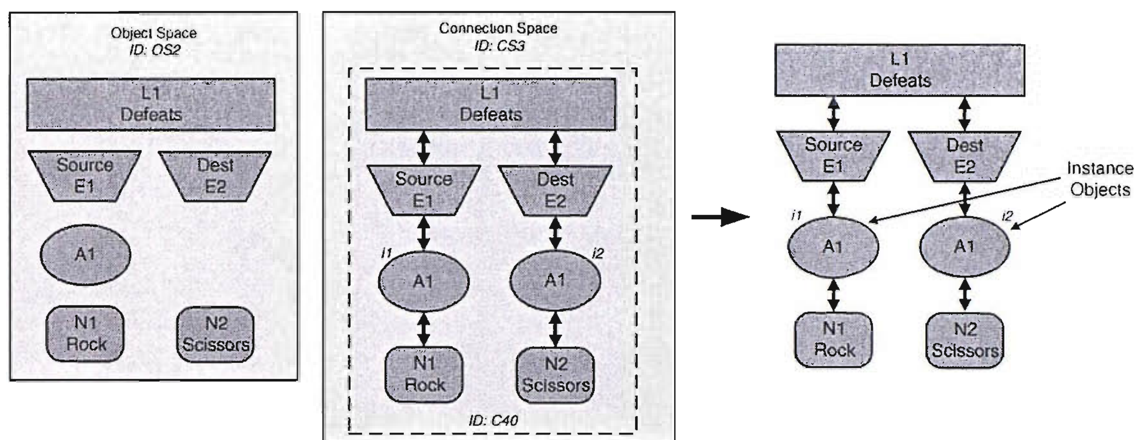


Figure 7.19: Instance Objects.

The use of Instance Objects is an efficient solution. No new Function Objects are created nor have any existing Function Objects had to be updated[8]. All instance information pertaining to a particular Instance Object is stored within the Connection Object within which it resides. This is because the different instances of a Function Object only exist within a Connection Object. This solution is akin to how re-use is carried out in Xanadu [Nelson 1999a] (Section 12.7.1).

7.11.2.2. Re-used Function Objects in External Connection Objects

The second problem is an extension of the first. How to associate a Function Object (within one Connection Object) with another Function Object re-used in multiple places within a second Connection Object?

Figure 7.20 shows an example where three Connection Objects are to be connected together as indicated by the arrows in the diagram. Connection Object C51 is to be conjoined to Connection Object C50 via the leftmost Function Object A1, and Connection Object C52 is to be conjoined to Connection Object C50 via the rightmost Function Object A1. The difficulty is how to associate each A1 Function Object in Connection Objects C51 and C52 with the correct corresponding A1 Function Object in Connection Object C50.

[8] This is relevant as regards the danger of object revision proliferation as explained in Chapter 9.

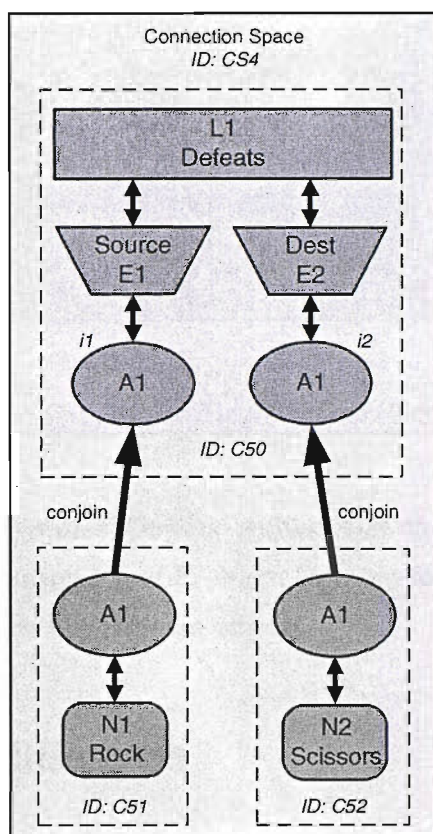


Figure 7.20: Attaching externally re-used Function Objects together.

Once again the solution adopted by the OCS data model is the use of Instance Objects. Function Objects that are external to Connection Objects containing re-used Function Objects can refer to re-used Function Objects using their Function Object identifier combined with that Connection Object's internal Instance Object identifier. Figure 7.21 shows the solution using Instance Objects for the problem represented within Figure 7.20.

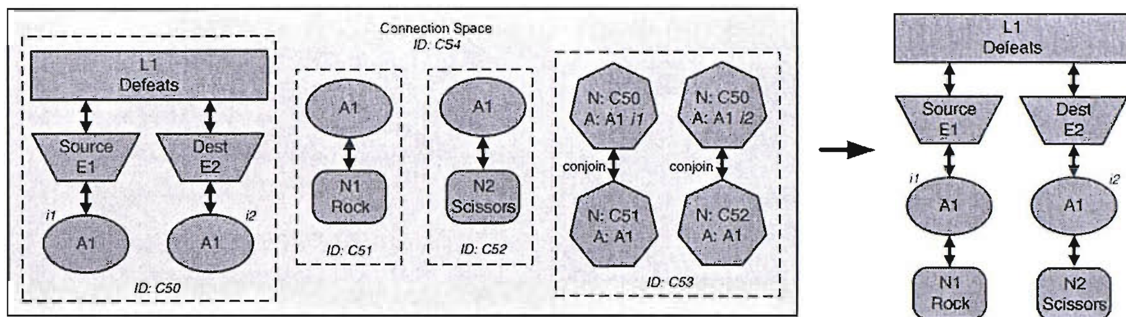


Figure 7.21: Assigning Instance identifiers to Connection Objects.

The implementation of Instance Objects shows that the OCS data model fulfils Computer Science Contribution 2 and Problem Domain Issue 1 – the ability to re-use the functionality of the same hypermedia objects.

7.12. XML Specification

Appendix A shows the XML representation for OCS Function and Connection Objects.

The appendix specification centres on the original Darmstadt OHP-Nav XML specification [Reich and Millard 1999]. This is to demonstrate how straightforward it is to adapt the four primary OHP-Nav objects (Node, Anchor, Endpoint and Link objects) for use within the context of the OCS data model.

The Darmstadt OHP-Nav XML specification has been chosen because not only is it one of the latest OHP-Nav specifications, but it is also a specification with which I am particularly familiar since it was the protocol used when I was involved with devising the inter-communication between components of the Solent CB-OHS [Reich et al. 1999b; Reich et al. 1999a].

Section A.2 shows the original XML specification for the four OHP-Nav primary objects whilst Section A.3 shows the XML specification of the same OHP-Nav object types adapted to the OCS Function Object format. This enables comparison between the two formats. Figure 7.22 shows an instance of an OHP-Nav Endpoint object in both formats.

```
<ENDPOINT>
    <ID> E1 </ID>
    <DIRECTION> source
    </DIRECTION>
    <ANCHORID> A1
    </ANCHORID>
</ENDPOINT>
```

Conventional OHP-Nav Endpoint.

```
<ENDPOINT>
    <ID> E1 </ID>
    <DIRECTION> source
    </DIRECTION>
</ENDPOINT>
```

OCS Endpoint Function Object.

Figure 7.22: Example of a conventional OHP-Nav Endpoint object and an OCS Endpoint Function Object (both expressed in XML format).

As can be seen, the only difference between the two object representations is that a Function Object does not record connection data unlike a conventional OHP-Nav object. Hence an OCS Function Object can continue to be used in the same functional manner as a conventional OHP-Nav object. The connection data is now recorded within OCS Connection Objects.

Section A.4 shows the XML representation for OCS Connection Objects. An XML example of an instance of an OCS Connection Object is shown in Figure 7.23. It reveals how Connection Object C51 of Figure 7.21 can be constructed.

```
<CONNECTION_OBJECT>
    <ID> C51 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
           <BOND> ATTACH </BOND>
           <OBJECT>
             <ID> A1 </ID>
           </OBJECT>
           <OBJECT>
             <ID> N1 </ID>
           </OBJECT>
        </CONNECTION>
```

```
    </CONNECTION_LIST>
</CONNECTION_OBJECT>
```

Figure 7.23: XML code to create Connection Object C51.

The XML tags work as follows:

The <CONNECTION_LIST> captures the connections that make up a Connection Object. The <CONNECTION> tags identify the individual connections between objects. The <BOND> tag describes whether an attach or conjoin operation is being used to connect the objects together. And finally the <OBJECT> tags specify the identity of the objects that are being connected together.

Appendix A also includes two further examples (in much greater detail) of XML code used to represent actual instantiations of OCS Function and Connection Objects. Section A.5 demonstrates how the same Function Objects can be re-used by multiple Connection Objects, and Section A.6 shows how a range of conventional OHP-Nav objects can be converted to the OCS data model format.

7.13. Class and Instance Relationship

On outward appearance, the relationship between Connection and Function Objects may be considered analogous to the relationship shared by a class and its instance objects[9]. But on closer inspection it becomes clear that this is not actually the case.

A class is essentially a template that describes the data and behaviour associated with objects (i.e. the instances) of that class. This is similar to the role of a Connection Object as it too acts as a template. In this case it instructs which Function Objects are to be connected together.

However a key difference is that a Connection Object does not guide which object *types* may be connected together. (This would be a Connection Object's role if it really were a class.) A Connection Object has no interest in what object types it is connecting together. This is left to the discretion of the Connection Object designer. For example a Connection Object could be used to connect an OHP-Nav Node to an

[9] A class instance object should not be confused with an OCS Instance Object (Section 7.11.2). The former is an object of a class whilst the latter is a virtual object (either Function or Connection Object) that is being re-used within a hypermedia structure.

Anchor (which is correct), or a Node to an Endpoint (which is incorrect). In either case, the Connection Object takes no interest in which arrangement is chosen. Moreover a Connection Object does not fulfil the true role of being a class since a Connection Object is an object in its own right with its own identity, and, like a Function Object, can be re-used (within other Connection Objects).

As regards Function Objects being compared with instance objects. Yes, Function Objects are instance objects, but they are instance objects of the object types they are used to describe, e.g. OHP-Nav Nodes, Anchors, etc. They are not instance objects of Connection Objects since a Function Object is not a type of Connection Object.

7.14. Object and Connection Space Example

This section demonstrates the re-use capability of the Object and Connection Space approach. It uses the "Rock, Paper, Scissors" game as an example (Section 7.6).

The hyperstructural relationship that 'Rock defeats Scissors' has already been created in Section 7.7 and is displayed in Figure 7.4. That same section also illustrates how that hyperstructure can be assigned to a Function Object Space (Figure 7.5) and a Connection Space (Figure 7.7).

On inspection of the Connection Space it would seem sensible to be able to re-use the 'Defeats' relationship represented by Connection Object C10 to express the other RPS relationships. Adoption of the OCS data model now makes this possible. Virtually all the objects and connections in the Object and Connection Spaces can be re-used to create the 'Paper defeats Rock' relationship (Figure 7.24).

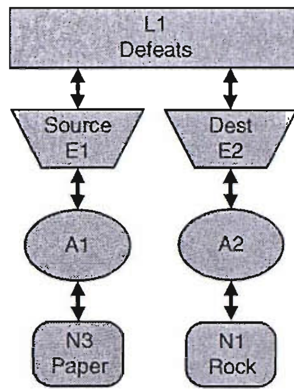


Figure 7.24: The 'Paper defeats Rock' relationship.

The overall arrangement of the Function Object and Connection Spaces necessary to represent both the 'Rock defeats Scissors ' and 'Paper defeats Rock' hyperstructures is shown in Figure 7.25.

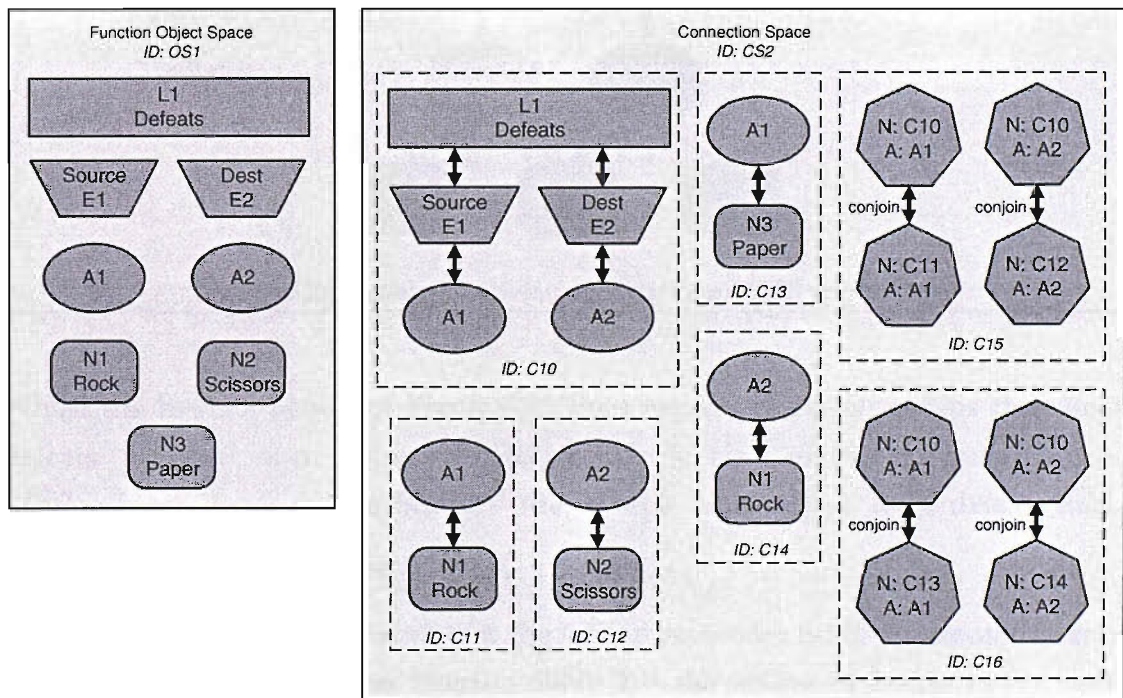


Figure 7.25: The Function Object and Connection Spaces necessary to represent 'Rock defeats Scissors ' and 'Paper defeats Rock' relationships.

As can be seen only 1 new Function Object (N3) and 3 new Connection Objects (C13, C14, and C16) have needed to be appended to the Object and Connection Spaces of

Figures 7.5 and 7.7 in order to create the 'Paper defeats Rock' hyperstructure relationship[10].

If the OCS data model were not adopted then it would not be possible to re-use the 'Defeats' hyperstructure as represented by Connection Object C10. Instead duplicate objects and connections would have to be created in order for that relationship to be re-used. This is wasteful of resources.

However, if the 'Defeats' relationship was re-used to represent both the 'Rock defeats Scissors' and 'Paper defeats Rock' relationships without the OCS data model then an incorrect hyperstructure would result. This is shown in Figure 7.26.

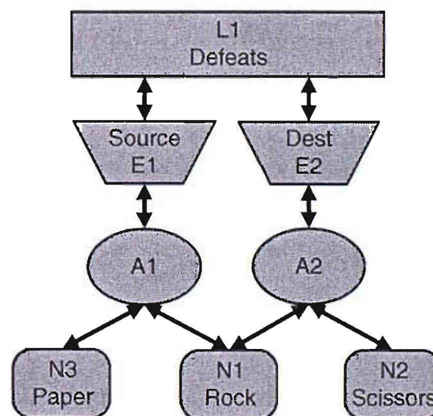


Figure 7.26: Incorrect hyperstructure without OCS approach.

Whilst the hyperstructure of Figure 7.26 does express the relationships that 'Rock defeats Scissors' and 'Paper defeats Rock', it also expresses some invalid relationships: 'Paper defeats Scissors' (the reverse is true) and 'Rock defeats Rock' (which is a draw).

This example has demonstrated that the OCS data model fulfils Computer Science Contribution 3 and Problem Domain Issue 2 – the ability to re-use hypermedia structure, i.e. inter-connected hypermedia objects. This has been possible through re-using Connection Objects which enable the same connections (as recorded by Connection Objects) to be re-used in the same or in different hypermedia structures. In the case of the example above the Connection Object being re-used is C10.

---

[10] It should be noted that Connection Object C15 has been added to Connection Space CS2. This Connection Object was not included in the original version of CS2 in Figure 7.7 since the node and anchor terminology of Connection Objects had not been introduced at that time.

7.15. Impact on the OHS Architecture

Through its low-level activities (Section 7.4) the OCS data model can benefit structure manipulation at all layers of the Open Hypermedia System architecture.

7.15.1. Client Application Layer

Client Applications can benefit in many ways when organizing hypermedia structure according to the OCS data model.

- *Easing Structure Creation.* Existing structures, in the form of Function and Connection Objects, can be re-used. This aids hypermedia structure creators as there is less 'new' structure to create which can be included as members of any structure that needs creating.

- *Structure Versioning and Hypertext Link Maintenance.* The impact that the OCS data model has on structure versioning and link maintenance forms the topic of Chapters 9 and 11 respectively.

- *Discovery of New Relationships.* New relationships can be discovered between the node resources that hypermedia structure is being used to connect. Table 7.1 shows how investigating the following Function Object types of an OHP-Nav hypertext link can reveal other potentially related node resources.

Object Type	New Relationships Discovered
Nodes	Identifies other hypertext links that reference the same node resource.
Anchors	Finds other hypertext links that reference the same content within any node resource. This is useful for locating node resources that share the same or similar content.
Endpoints	Identifies other hypertext links of the same direction.
Links	Finds relationships (i.e. links) of a certain type.

Table 7.1: Discovering new node object resource relationships.

Such relationships are easier to find when using the OCS approach, because the same Function Object may be being re-used within different Connection Objects (i.e. different relationships between node resource objects). Therefore it is only

necessary to search for that Function Object's identifier within other Connection Objects in order to re-create (and hence identify) other related hypertext links.

- *Reduced Message Sizes*. The size of messages dispatched between Structure Server and Client Application can potentially be reduced. Fewer Function Objects need to be sent between components if the same items are being re-used within the structure being transmitted.

7.15.2. Middleware Layer

The notion of separating hypermedia connections from hypermedia objects makes the OCS data model a re-usable concept as the OCS approach can be adopted by any hypermedia server regardless of the domain they serve.

The OCS data model also assists hypermedia domain interoperability as it enables the hypermedia objects of one domain to be directly re-used within other domains. Individual hypermedia objects (or blocks of inter-connected objects) can be re-used without also having to re-use the other objects to which they may be connected within other structures (of possibly other domains). This is a particularly useful facility for FOHM as its data model promotes the notion of utilising the same structures within different hypermedia domains.

7.15.3. Storage Back End Layer

The OCS data model offers optimization at the storage level. The re-use of hypermedia objects should mean fewer objects to store in the Storage Back End. This not only benefits the Storage Back End as fewer objects need storage, but it also benefits the maintainers of the Storage Back End as fewer objects need management.

7.16. Added Complexity

For all its benefits, there is no avoiding the fact that the OCS data model does bring added complexity to the organisation of hypermedia structure. Notably, hypermedia objects must now be allocated to separate Function and Connection Spaces. (The available options are discussed in Section 7.16.2.) But it is this very act of separation that brings the many advantages (Section 7.15).

The process of separating structures can be compared to the added complexity when operating an independent linking scheme (e.g. OHP-Nav) versus operating a much simplified link embedding scheme (e.g. HTML of the World Wide Web). This is because the independent link approach requires more components to administer: Client Applications, Hypermedia Servers and Storage Back Ends. Moreover there are a greater number of messages to send between components to carry out hypermedia functions. The result is greater time spent when creating and following links.

But adopting independent links also offers many advantages. Noticeably being able to re-use the same Client Applications with multiple OHSs, and being able to re-use the same links with different documents (e.g. Microcosm generic links). Therefore despite the added complexity it is oft argued that the advantages of operating an independent linking scheme outweigh the disadvantages.

The same can also be concluded about the OCS data model. Whilst it brings added complexity, it also brings many advantages. Prominent are the re-use of hypermedia objects, collections of connected objects, improved structure versioning and improved link maintenance.

Chapter 8 explores the implications of applying the OCS data model to OHP-Nav and FOHM data models. In particular Chapter 8 discusses how the predicted additional complexity impacts upon performance in terms of the extra number of hypermedia objects generated and saved as a result of organising hypermedia objects according to the Object and Connection Space approach.

What follows is a list of the major added complexity that the OCS data model introduces when manipulating structure in the general case.

7.16.1. Separating and Building Structure

Adoption of the OCS data model adds the complication that hypermedia structure must be separated into Function and Connection Objects. However some of this complexity can be reduced if the structure is organised into connection arrangements that imitate an existing open hypermedia data model's conventional connection arrangements, e.g. OHP-Nav. This can make users already familiar with an existing open hypermedia structure data model more at ease with the OCS data model. Section 8.5 explains how the OCS data model can be used to emulate OHP-Nav connection patterns.

Once the structure has been separated into Function and Connection Objects, there is the added complexity that that same structure then has to be re-built to enable Client Applications to use it in some fashion, e.g. for traversal. Section 8.5 also describes how the OCS data model retrieval process can mimic the OHP-Nav re-building of structure.

7.16.2. Re-using Structure

One of the components of the OHS architecture (Client Application, Hypermedia Server or Storage Back End) has to determine which Function Objects (of a hypermedia structure) it is possible to re-use. The three choices are:

- *Client Application*. The client can determine which structure to re-use as and when it creates the structure.
- *Hypermedia Server*. The Hypermedia Server can determine which structure is re-usable when the Client Application sends the structure for storage via the Hypermedia Server.
- *Storage Back End*. Once the structure has been sent for storage, the Storage Back End component can then determine which structure segments are identical and can be re-used.

There is then the added complexity that once a Function Object or Connection Object has been re-used, it will be necessary to uniquely identify each different re-use instance of it. This is to ensure that the correct re-used Function or Connection Object is referenced by other structural objects. However this added complexity has been solved by the introduction of Instance Objects as described in Section 7.11.2.

7.17. Summary

This chapter has introduced the Object and Connection Space data model. The main aim of which has been to open up hypermedia structure. This has been accomplished through the separation of hypermedia objects and connections.

Critical to the OCS data model are the introduction of two special object types: Function and Connection Objects. Function Objects are conventional hypermedia objects minus their connection data, and Connection Objects describe the connections between Function Objects. Through combination of these two object types, re-use of

the same hypermedia objects and connections is possible in order to create re-usable hypermedia structures. Such structures can then be used to forge new relationships between node resource objects of a hypertext network.

The chapter also points out how the OCS data model fulfils Computer Science Contributions 1, 2 and 3 (of Chapter 1), and Problem Domain Issues 1 and 2 (of Chapter 6). But a majority of the chapter has focused on the underlying mechanisms of the OCS data model. This has included explaining how Connection Objects connect Function and/or Connection Objects together. A further issue has been anchoring within Function Objects, i.e. how to reference a specific Function Object within a given Connection Object. And a final capability under examination has been how to uniquely identify re-used Function Objects within a given hyperstructure. This is the cornerstone of being able to build hyperstructure composed of re-used objects.

Chapter 8. Implications for OHP-Nav and FOHM

8.1. Introduction

This chapter explores the implications of applying the Object and Connection Space approach to the OHP-Nav and FOHM data models previously described in Chapter 3. It shows how through converting the organisation of OHP-Nav and FOHM structure to Function and Connection Objects, the OCS data model satisfies Computer Science Contributions 1 and 2 (of Chapter 1), and resolves Problem Domain Issues 1 and 2 (of Chapter 6).

Restated they are:

CSC1:	Extending the concept of open hypermedia into the realm of hypermedia structure.

CSC2:	Promoting the general re-use of hypermedia structure.

PD1:	The use and storage of hypermedia objects that carry out the same function.

PD2:	The use and storage of identical hypermedia structure.

The contention of this chapter is that the conventional organisation of structure within OHP-Nav and FOHM unnecessarily restricts the opportunity for hypermedia object and connection re-use. This is contrasted with the application of the OCS data model which by comparison facilitates greater hypermedia structure re-use (CSC2).

A further benefit brought about by the OCS data model is that fewer hypermedia objects are created. This has the knock-on effect that fewer objects need to be stored by the hypermedia system, subsequently less hypermedia object management will be deemed necessary.

8.2. Embedding within Hypermedia Objects

The main reason why the implementation of both the OHP-Nav and FOHM data models are inefficient is because they embed hypermedia object references within the hypermedia objects themselves. This is in the linked-list fashion as described in Sections 3.2.5 and 3.3.4:

- *OHP-Nav data model.* Link objects store embedded references to Endpoints; Endpoints store embedded references to Anchors; and Anchors store embedded references to Nodes.

- *FOHM data model.* Bindings store embedded references to Reference objects; and Reference objects store embedded references to Data objects or Association objects.

The two data models are inefficient because they prevent the re-use of single objects and the re-use of structure segments (Sections 6.2 and 6.3). This occurs for two reasons:

1. Hypermedia objects that would otherwise be functionally identical may have different content because they are embedded with different object reference identifiers within their content. The result is that one functionally identical object cannot be swapped for another.

2. Embedding object references means that each OHP-Nav or FOHM object can only be re-used in association with the objects that it stores the references to. Thus if an object contains a reference to a second object, then the first object can only be re-used in conjunction with the referenced object. Hence the first object cannot be re-used in isolation as a single entity. Moreover, hypermedia structure designers will often be prevented from creating structure segments comprising the objects of their choice. This is because a structure segment, in addition to containing the existing objects of the segment, must also contain the secondary objects referenced by the existing objects of the segment. This is whether these extraneous secondary objects are meant to be members of the segment or not.

The FOHM data model further inhibits hypermedia object and connection re-use because it permits wholesale object embedding, i.e. one FOHM object being entirely contained within another FOHM object. The problem is that this prevents the re-use of embedded objects as separate entities because any re-use of the embedded object would also necessitate re-use of the container object. Wholesale object embedding

also prevents the re-use of container objects as separate entities since when the container object is re-used, the embedded object within it will also automatically be re-used as well.

8.3. Application of the OCS Data Model

The role of the OCS data model is to re-organise the functional and connectional data within hypermedia objects. This is achieved by assigning hypermedia objects (minus their embedded object references) to Function Objects, and transforming the embedded object reference data to become Connection Objects. This results in embedded object reference data no longer forming part of the content of hypermedia objects. Such separation enables the OCS data model to facilitate hypermedia object and connection re-use. Hence the issue of re-using single objects and segments of hypermedia structure is resolved:

1. Removing object reference identifiers from being embedded within individual hypermedia objects means functionally identical objects can now share the same content. Thus functionally identical objects can be re-used in place of one another.

2. Hypermedia structure designers are no longer forced to re-use all the objects that would otherwise be pointed at by embedded reference identifiers. Instead structure designers can use OCS Connection Objects to pick and choose which objects and/or connections to re-use.

As regards FOHM wholesale object embedding, the OCS data model leaves the decision on whether to embed whole objects within one another as a design choice for OCS hypermedia structure creators. This is because it may be the case that there is no benefit to be gained from separating an embedded object from its containing object. For example, if both the embedded object and its container are only ever re-used as a pair then no advantage is gained by separating the two objects whereby an additional object (i.e. the embedded object as an independent object) is created that requires additional object management and storage.

However, as a rule, the OCS data model does not encourage wholesale object embedding since as already explained in Section 8.2, a greater range of re-use opportunities exist if the embedded object and its container are separated from one another.

8.4. Resolving Problem Domain Issues 1 and 2

In defining Problem Domain Issues 1 and 2, Sections 6.2 and 6.3 outline four particular problem scenarios where the OHP-Nav and FOHM embedded object reference techniques hamper the re-use of individual objects and whole segments of hypermedia structure. This section describes how adopting the OCS data model resolves two of those problem scenarios. These are: enabling the re-use of OHP-Nav single objects (Section 8.4.1) and enabling the re-use of FOHM structure segments (Section 8.4.2).

Appendix B describes how the OCS data model resolves the remaining two scenarios.

8.4.1. OHP-Nav Single Hypermedia Object Re-use

Adopting the OCS approach enables re-use of the 15 functionally identical objects within the three hypertext links of Figure 6.1 of Section 6.2.1. Figure 8.1 shows an example OCS solution, and Table 8.1 presents a summary of the object re-use taking place.

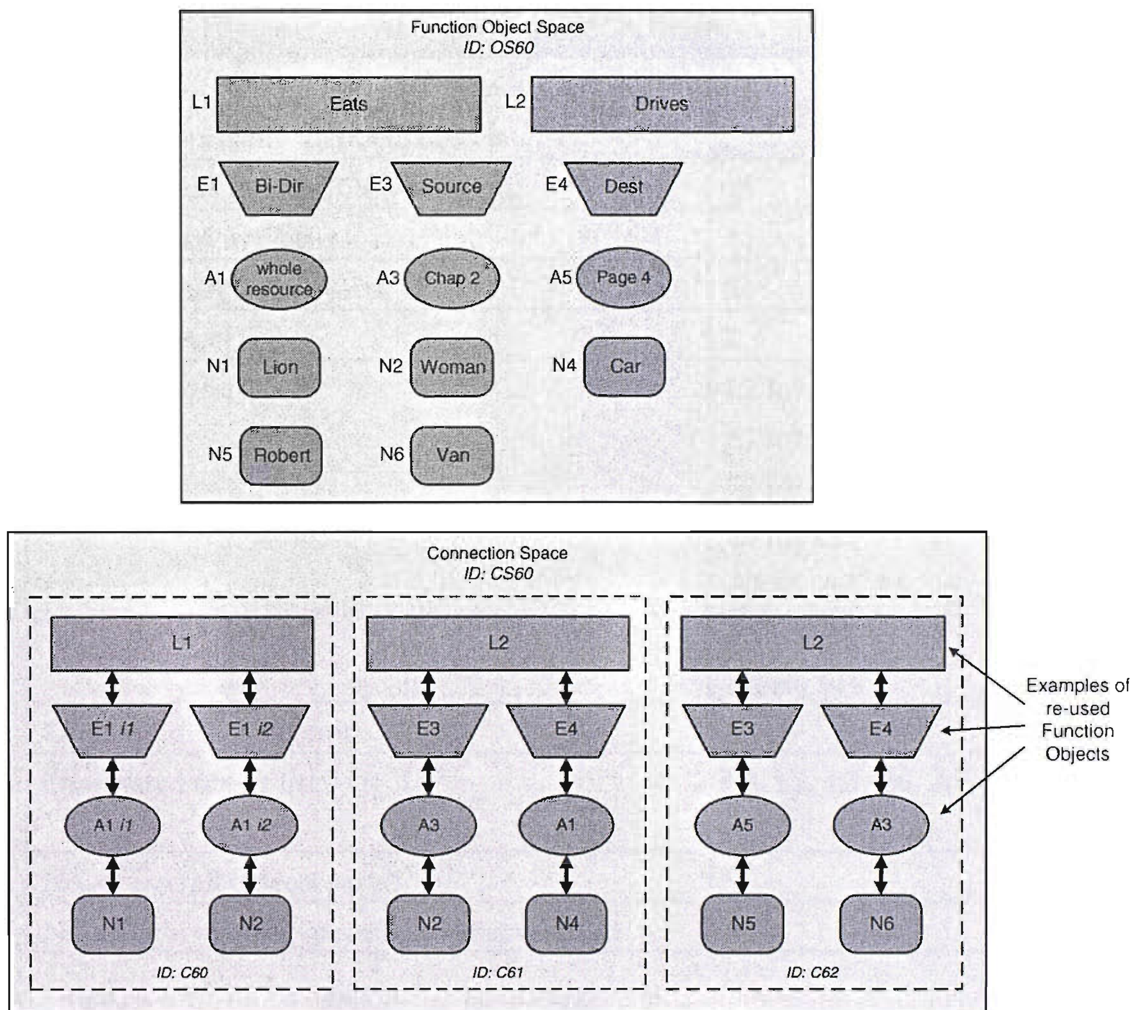


Figure 8.1: OCS solution enabling individual OHP-Nav hypermedia object re-use for the hypertext link structures of Figure 6.1.

The three Connection Objects C60, C61 and C62 build the three hypertext links of Figure 6.1. Thanks to the OCS data model, it is now possible to re-use 7 single objects whereas before (Section 6.2.1) it was only possible to re-use 1 single object.

Description	Value
No. of objects to build original hypertext links	21
No. of functionally identical objects	15
No. of objects to build OCS solution	16
No. of Function Objects	13
No. of Connection Objects	3
Re-used object count	7
Re-used object list	L2 for L3 E1 for E2 E3 for E5 E4 for E6 A1 for A2, A4 A3 for A6 N2 for N3
Eliminated object count	8
Eliminated object list	L3, E2, E5, E6, A2, A4, A6, N3
No. of overall objects saved	5
No. of equivalent hypermedia objects saved	8

Table 8.1: Summary of the re-use taking place within the OCS solution of Figure 8.1.

Table 8.1 shows that the OCS data model solution uses a total of 16 objects. This compares with a total of 21 objects used by the OHP-Nav object reference embedding approach. Hence the OCS solution saves 5 objects.

However, of the 16 OCS objects produced, only 13 are Function Objects. It is here where the real comparison between the OCS approach and the conventional approach should take place. The OCS Function Objects are the heavyweight objects (Section 7.9) which contain actual hypermedia object information and are identical to the hypermedia objects produced by the conventional approach except they do not contain embedded connection information. The remaining 3 OCS objects are Connection Objects. These are lightweight objects which do not contain a great amount of data. Therefore the true comparison is between the 13 objects produced by the OCS approach versus the 21 hypermedia objects generated by the conventional approach.

In this way the OCS solution actually achieves a greater saving of 8 equivalent hypermedia objects compared with the original OHP-Nav object reference

embedding approach. Therefore due to the OCS data model it has been possible to eliminate 8 hypermedia objects from storage.

8.4.2. FOHM Repetitive Hypermedia Structure Re-use

Section 6.3.2 explains how the FOHM data model prevents the re-use of the identical FOHM structure segment within Figure 6.6. The goal is to re-use the first highlighted segment twice within the same hypermedia structure so that the highlighted segment is not only used at its original location but also in place of the second highlighted segment.

Whilst not possible using the FOHM embedded object reference approach, such re-use is possible when adopting the OCS data model. The OCS solution is shown in Figure 8.2 along with Table 8.2 which summarises the object re-use taking place.

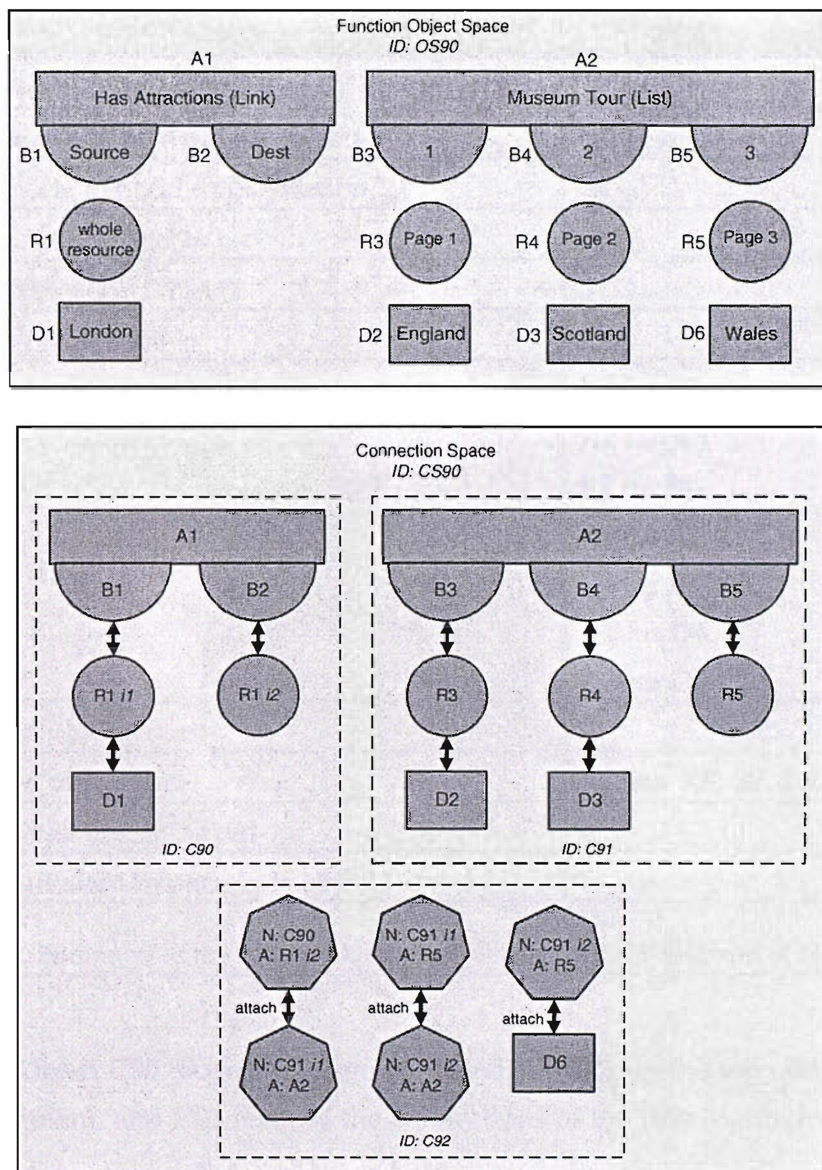


Figure 8.2: OCS solution enabling the same structure segment to be re-used twice within the same FOHM structure of Figure 6.6.

The OCS solution shown in Figure 8.2 permits wholesale object embedding. This is because no object re-use benefit is gained by separating any of the embedded objects from their containers since neither an embedded object or container of an embedded object needs to be re-used as a single object.

Description	Value
No. of objects to build original FOHM structure	17
No. of functionally identical objects	14
No. of objects to build OCS solution	13
No. of Function Objects	10
No. of Connection Objects	3
Re-used object count	7
Re-used object list	R1 for R2 A2 for A3 R3 for R6 R4 for R7 R5 for R8 D2 for D4 D3 for D5
Eliminated object count	7
Eliminated object list	R2, A3, R6, R7, R8, D4, D5
No. of overall objects saved	4
No. of equivalent hypermedia objects saved	7

Table 8.2: Summary of the re-use taking place within the OCS solution of Figure 8.2.

Connection Object C90 records the connections that make up the top non-highlighted structure segment, and C91 records the connections of the first highlighted structure segment.

C92 is the Connection Object responsible for building the whole hypermedia structure. It re-uses Connection Object C91 twice. Each use of the Connection Object is assigned a different instance identifier in order to denote that C91 is being re-used in different places within the same Connection Object (i.e. within the same structure). The first instance, marked as *i1*, is used to build the first highlighted segment, and the second instance, marked as *i2*, is used to build the second highlighted segment.

As Table 8.2 shows, this OCS solution uses a total of 13 objects. This compares with the 17 objects needed to build the FOHM structure using the conventional FOHM object reference embedding approach (Figure 6.6). Thus the OCS solution saves a total of 4 objects. However, because 3 of the OCS objects are Connection Objects (deemed to be lightweight objects), then the real saving amounts to 7 equivalent hypermedia objects.

Figure 8.3 displays the hypermedia structure produced by Connection Object C92. Of relevance is that the second highlighted segment contains the same objects and connections as the first highlighted segment.

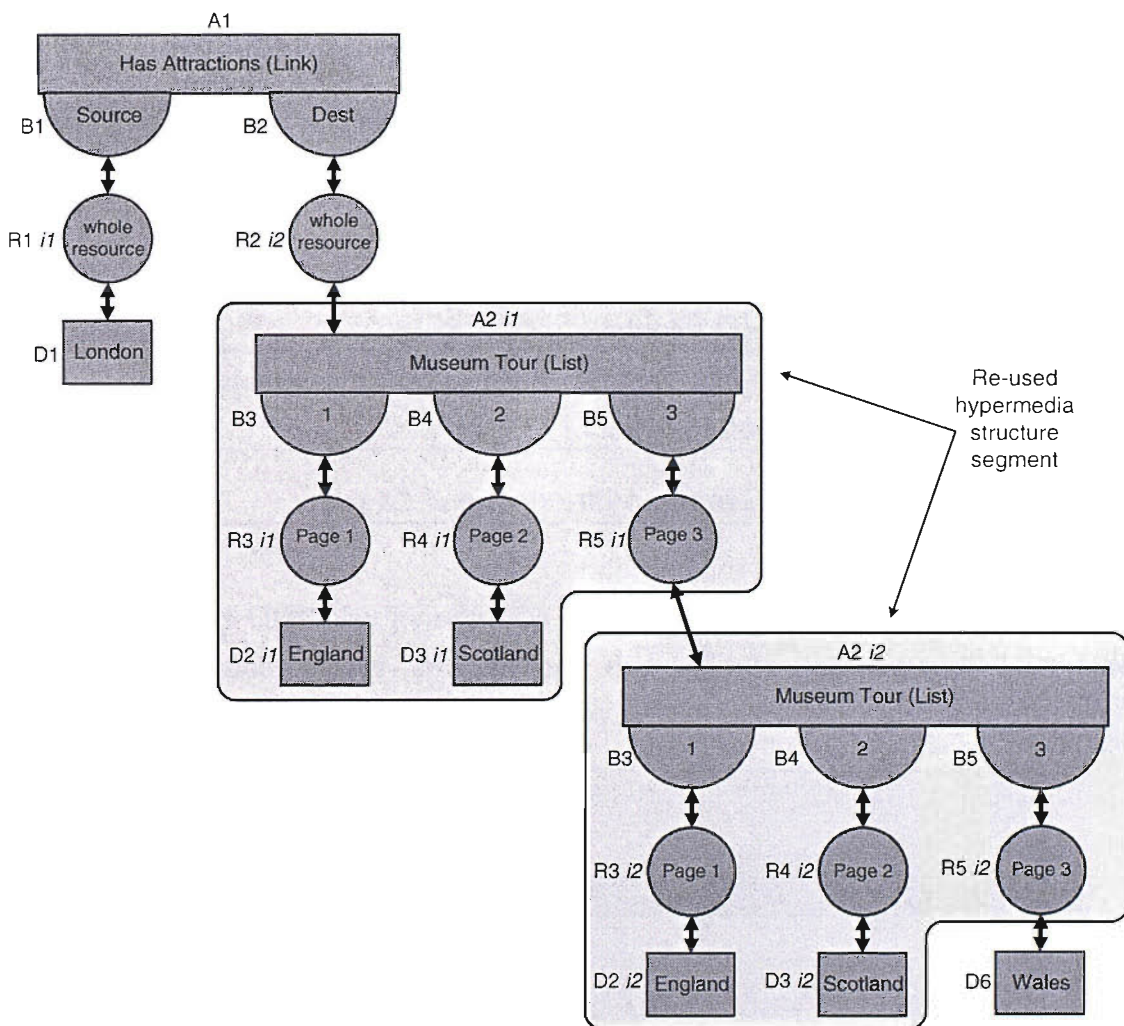


Figure 8.3: FOHM hyperstructure containing a re-used structure segment.

8.5. Imitating OHP-Nav Structural Organisation

As pointed out in Section 7.16 adopting the OCS data model brings the added complication that hypermedia structure must now be divided into Function and Connection Objects. That same section also points out that some of this complexity can be reduced by organising OCS structure into connection arrangements that imitate an existing open hypermedia data model's conventional connection arrangements. The intention being to make users already familiar with the existing

hypermedia data model feel more at ease with the OCS data model representation. Such imitation can be achieved, for example, by modelling OCS object-connection relationships on OHP-Nav object-connection relationships. Table 8.3 shows how this is possible.

OHP-Nav Relationship	OCS Data Model Relationship
Anchor stores reference to single Node.	Connection Object connecting single Anchor to single Node.
Endpoint stores reference to single Anchor.	Connection Object connecting single Endpoint to single Anchor.
Link stores references to one or more Endpoints.	Connection Object connecting single Link to multiple Endpoints.

Table 8.3: Imitating OHP-Nav relationships.

Figure 8.4 shows OHP-Nav hypermedia structure 'Rock defeats Scissors' organised into a Connection Object arrangement that mimics conventional OHP-Nav connections.

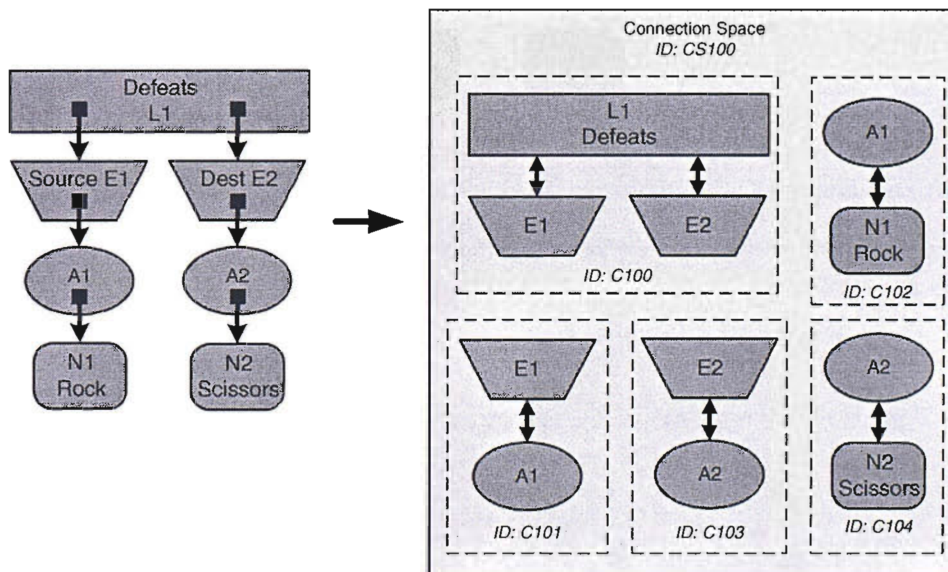


Figure 8.4: Imitating OHP-Nav hypermedia structure representation.

When it comes to re-building structure, because OHP-Nav embeds connection data in individual hypermedia objects, then for every Function Object retrieved, its associated Connection Object will also have to be retrieved as well. For example to rebuild structural segment E2 to A2 (of Figure 8.4), then when Function Object

Endpoint E2 is retrieved so will Connection Object C4 in order to determine that E2 is connected to A2.

8.6. Applying the OCS Data Model to Connection Objects

This section explores whether it is worthwhile applying the OCS data model to OCS Connection Objects. This is because, like OHP-Nav and FOHM structures, OCS Connection Objects are also comprised of objects and connections. This means it is possible to apply the OCS data model to individual Connection Objects. Figures 8.5 and 8.6 highlight the similarities between an OHP-Nav structure and an OCS Connection Object.

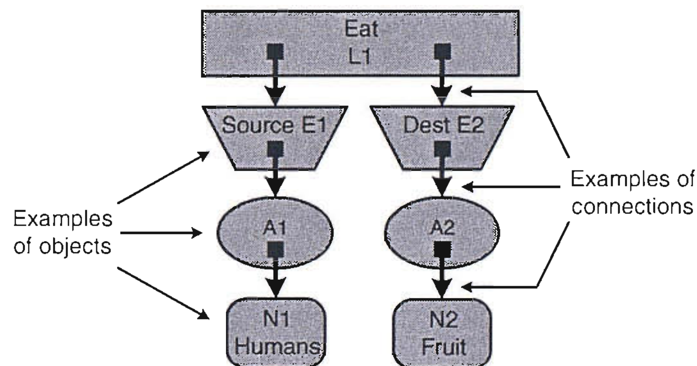


Figure 8.5: An OHP-Nav hypermedia structure expressing the 'Humans Eat Fruit' relationship.

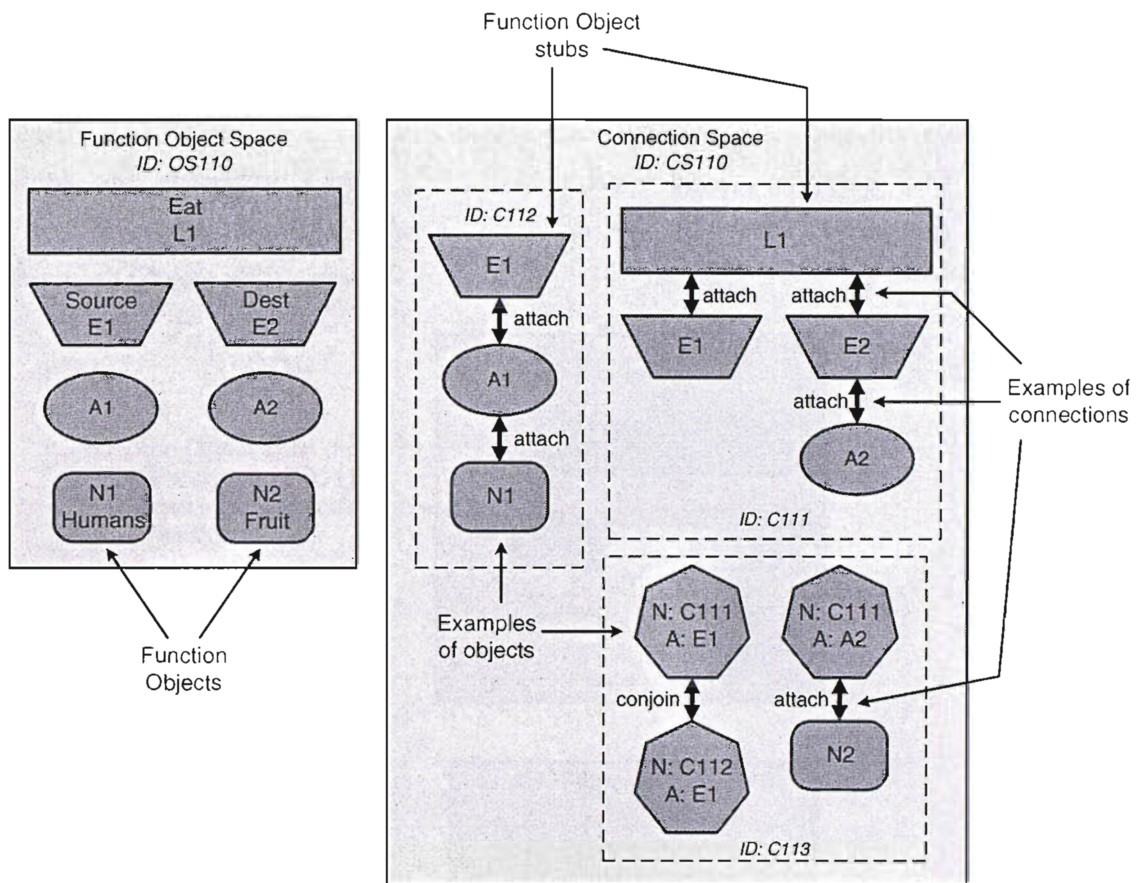


Figure 8.6: An example of how the elements of the structure of Figure 8.5 can be assigned to Object and Connection Spaces via the OCS data model.

In question is whether the same advantages afforded to OHP-Nav and FOHM structures can also be afforded to OCS Connection Objects. I.e. can the internal objects and connections within Connection Objects be re-used in order to prevent the wasteful storage of unnecessary object and connection entities?

In order to apply the OCS data model to Connection Objects, they must first be represented as FOHM structures. Subsections 8.6.1, 8.6.2 and 8.6.3 explain why this FOHM transformation is necessary. This is followed by Subsection 8.6.4 which explains what (if any) benefits are gained from applying the OCS data model to Connection Objects.

8.6.1. Why Change the Structure Representation of Connection Objects?

The reason why the OCS data model should not be applied directly to Connection Objects (without first being converted to a FOHM structure) is because such action

does not produce any meaningful object re-use opportunities. Figure 8.7 shows an example where the OCS data model is applied directly to Connection Object C111 (of Figure 8.6). It shows Function Object Space OS120 containing the Function Object stubs of Connection Object C111 which have been converted into independent Function Object entities, and Connection Space CS120 containing an example Function Object arrangement.

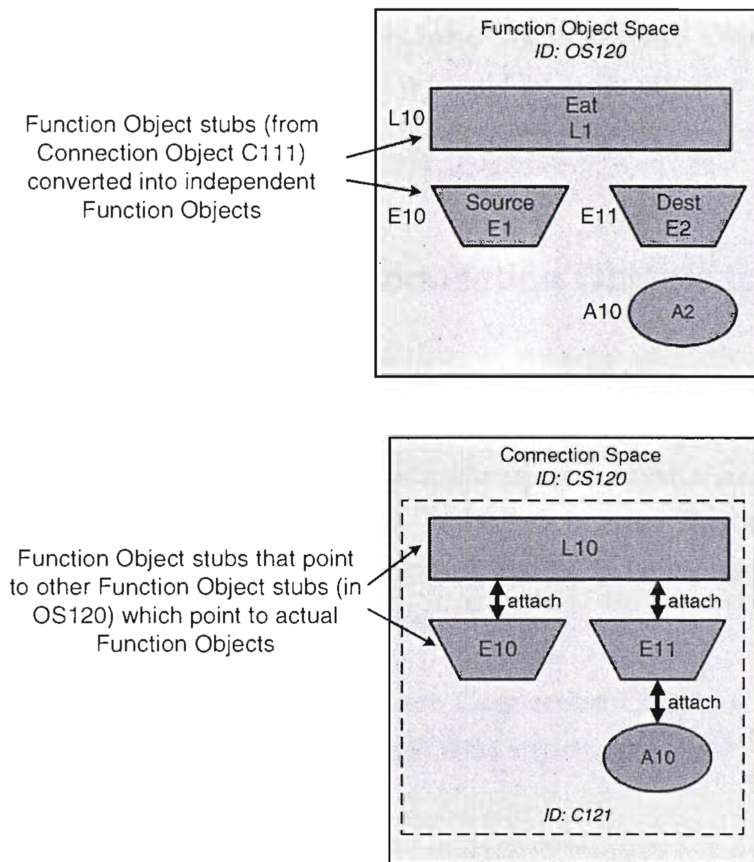


Figure 8.7: Ineffectual application of the OCS data model to Connection Object C111 of Figure 8.6.

Applying the OCS data model directly to Connection Object C111 is essentially a pointless exercise because the object re-organisation taking place within new Connection Space CS120 could equally and more efficiently be carried out directly within original Connection Object C111. This is for two reasons:

1. The only role that each Function Object of OS120 plays is to act as a pointer to the Function Objects of original Function Object Space OS110, e.g. Link object L10 in OS120 acts as a stub for L1 in OS110. Hence the Function Objects of OS120 are redundant creations which lead to unnecessary object management and storage. Therefore new Function Object Space OS120 and the Function Objects within it

need not exist. It would be more efficient if Connection Object C121 referred directly to the Function Objects of OS110 instead.

2. New Connection Space CS120 and the Connection Objects within it also need not exist. This is because CS120 only has access to the same objects (to re-organise) as original Connection Object C111. This is because each Function Object within OS120 acts as a pointer to a corresponding Function Object in OS110. Thus CS120 only has opportunity to organise the Function Object stubs in the same formations as C111. Therefore it would be more efficient to leave C111 to do such object organising, and not bother creating Connection Space CS120 in the first place.

8.6.2. Transforming Connection Objects to FOHM Structures

The reason why Connection Objects are converted to FOHM structures is to reveal the object and connection data that make up the connections between each Function Object stub. Hence it is to these FOHM structures that the OCS data model is applied. Any re-use that takes place via the OCS data model is between the objects and connections within these FOHM structures.

Figure 8.8 shows an example where Connection Objects C111, C112 and C113 (of Figure 8.6) have been converted to FOHM structures.

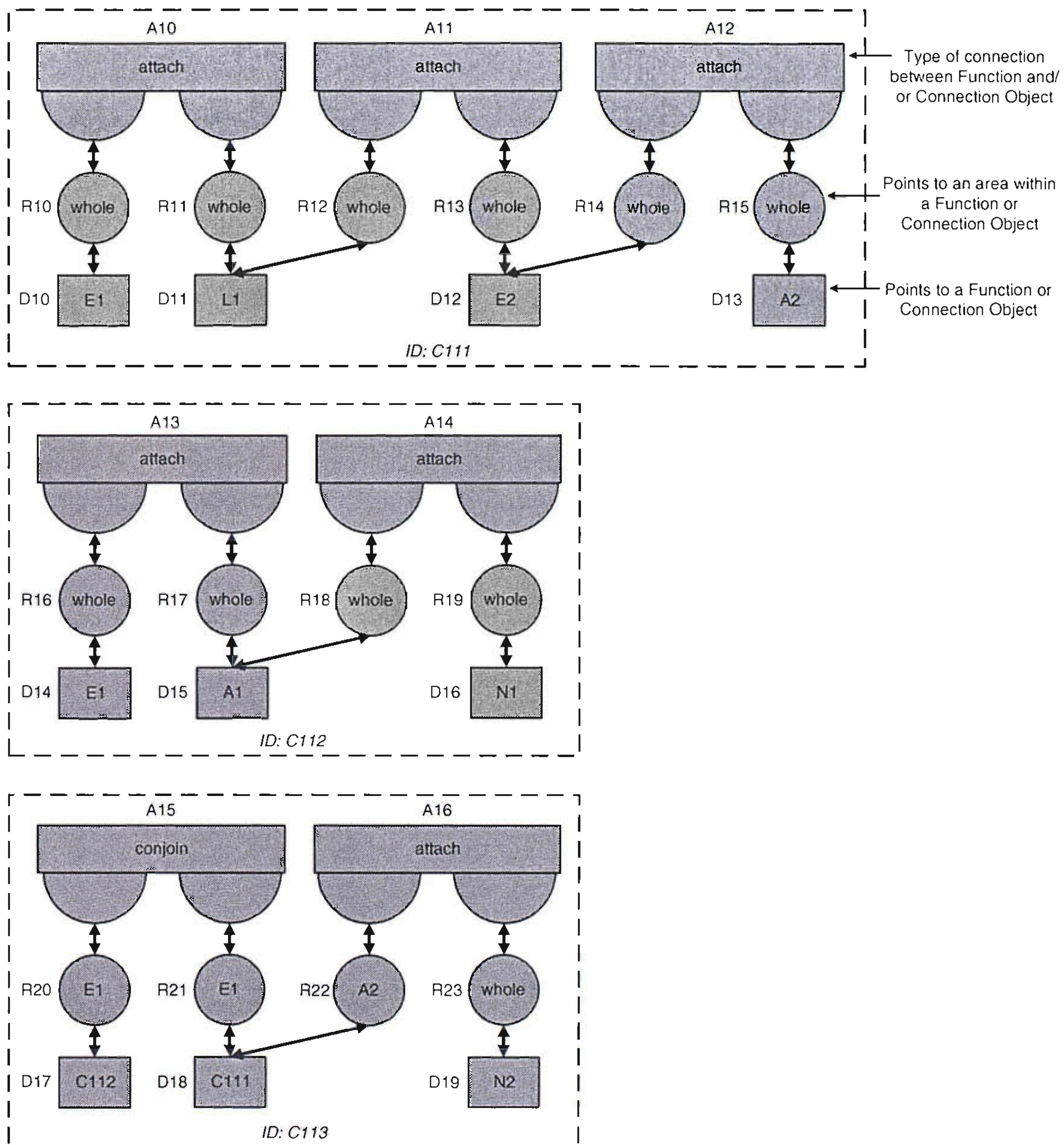


Figure 8.8: Transformation of Connection Objects C111, C112 and C113 to FOHM structures.

Each FOHM object within a transformed Connection Object has the following role:

- *Data objects*. Each Data object (within a Connection Object) corresponds to a Function or Connection Object stub. For example Data object D10 (in Connection Object C111) has content E1 which means it corresponds to Function Object E1.

- *Association objects*. Each FOHM Association object represents a single connection. In Connection Object C111 there are three Association objects which means that there are a total of three connections within this Connection Object. One example of a connection is A11 attaching D11 to D12 which signifies that Function Link

object L1 is connected to Endpoint object E2. A FOHM Association object also denotes the type of relationship between each connected Function and/or Connection Object, i.e. they may be *conjoin*ed (Section 7.10.1) or *attach*ed (Section 7.10.2).

- *Reference objects.* When needing to reference a Function Object within an existing Connection Object, the ID of that Function Object is recorded within a FOHM Reference object. For example in Connection Object C113, Node N2 wants to attach to Anchor A2 which is located inside Connection Object C111. Therefore A2 is recorded as the Reference value within R22 of 'attach' Association A16.

- *Binding objects.* Their role is to form the bridge between a Reference and Association object. They have a valueless state since neither Function nor Connection Objects are attached or conjoined to one another in any particular order.

8.6.3. Applying the OCS Data Model to FOHM Structures

Once an OCS Connection Object is converted to a FOHM structure, the OCS data model can then be applied to the internal connections within the Connection Object. Figure 8.9 shows an example of applying the OCS data model to the three Connection Objects of Figure 8.8.

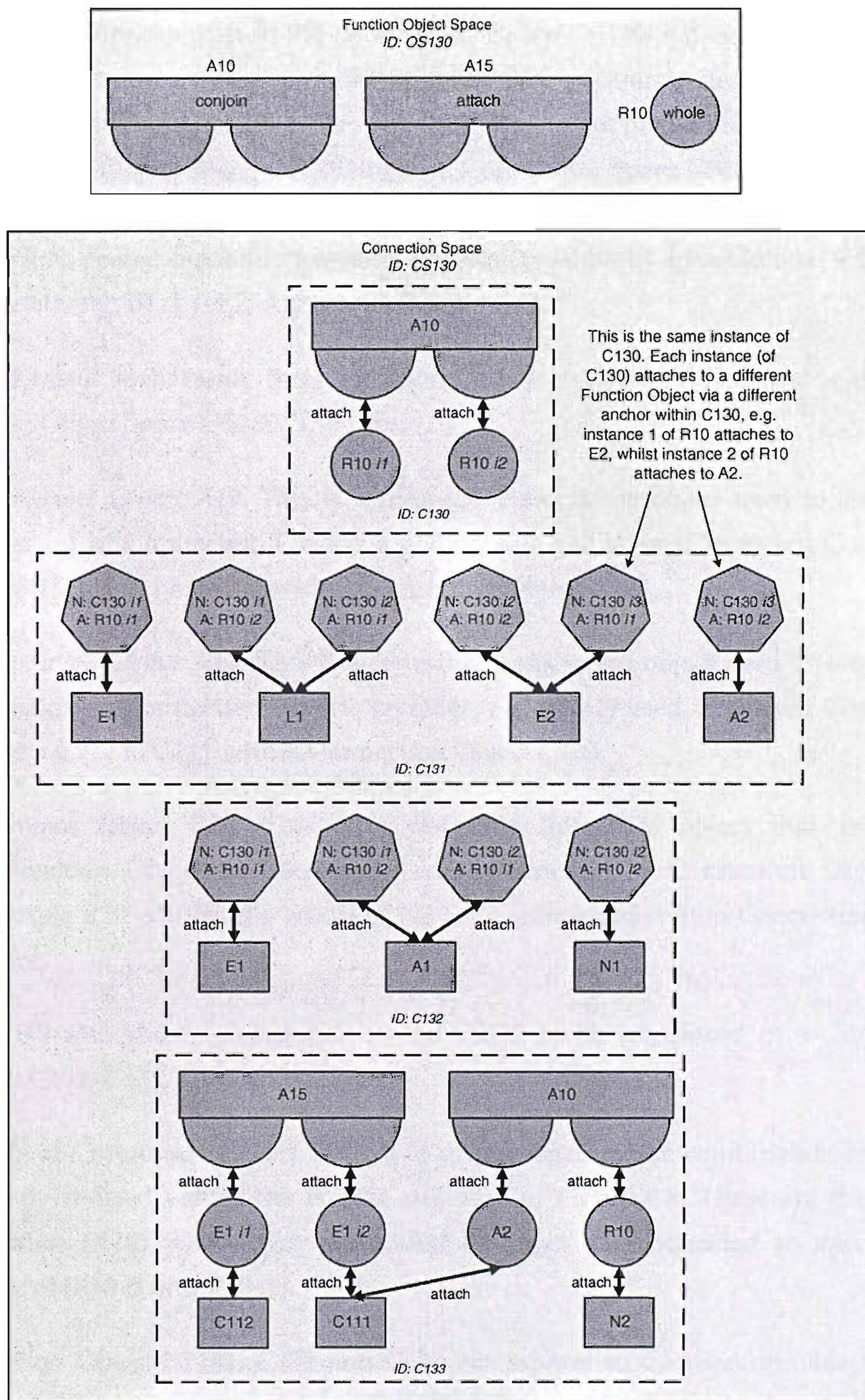


Figure 8.9: An example OCS solution having applied the OCS data model to the Connection Objects of Connection Space CS110 of Figure 8.6.

Figure 8.9 shows that a significant number of Function Objects referenced within the Connection Objects of Connection Space CS130 are not recorded within Function Object Space OS130, e.g. objects E1, L1 and C112. The reason for this is because all

these objects already exist in Function Object Space OS110 and/or Connection Space CS110. They were created as part of the earlier OCS solution to describe 'Humans Eat Fruit' shown in Figure 8.6. Therefore those objects do not have to be re-created within new Function Object Space OS130. Instead, Connection Space CS130 can simply pick and choose which Function and/or Connection Objects to re-use from OS110 or CS110. Such re-use capability prevents needless creation of an additional 9 Function Objects: N1, A1, E1, L1, E2, A2, N2, C111 and C112.

The net result is that only three Function Objects need to be recorded within new Function Object Space OS130. They are:

- *Association Object A10*. This is a re-usable Association object used to *attach* two Function or Connection Objects together, e.g. A10 is used to attach Connection Object C111 to Node N2 within Connection Object C133.

- *Association Object A15*. This is a re-usable Association object used to *conjoin* two Function or Connection Objects together, e.g. A11 is used to conjoin Connection Object C112 to C111 within Connection Object C133.

- *Reference Object R10*. This is a re-usable Reference object that enables a Connection Object to reference a whole Function or Connection Object. For example R10 enables the whole of N2 to be referenced within Connection Object C133.

Figure 8.9 also shows Connection Space CS130 to be comprised of 4 Connection Objects: C130, C131, C132 and C133.

The role of Connection Object C130 is to group together the combination of objects frequently re-used within the FOHM structure of Figure 8.8. These are the 'attach' Association (A10) whose two embedded Bindings are connected to two 'whole' References (R10 *i1* and R10 *i2*).

Connection Objects C131, C132 and C133 correspond to Connection Objects C111, C112 and C113 of Connection Space CS110 (of Figure 8.6). As can be seen Connection Objects C131 and C132 re-use C130 10 times. Such re-use prevents creation of 13 additional Function Objects (R11, A11, R12, R13, A12, R14, R15, A13, R16, R17, A14, R18 and R19). This is because every re-use of C130 prevents the creation of an extra 3 Function Objects (equivalent to A10 and 2 instances of R10) that would otherwise have to be recorded in Function Object Space OS130.

Figure 8.8 shows the total number of objects that would be needed to create the FOHM structure representation of the three Connection Objects C111, C112 and C113 if no re-use was possible. But adopting the OCS solution to enable re-use within Connection Objects C131, C132 and C133 (of Figure 8.9) means that the only objects that need to be recorded in Function Object Space OS130 are A10, A15 and R10. Thus the following objects (from Figure 8.8) need not be created: D10, R11, D11, A11, R12, R13, D12, A12, R14, R15, D13, A13, R16, D14, R17, D15, A14, R18, R19, D16, R20, D17, R21, D18, A16, R22, R23 and D19.

Description	Value
No. of objects to build FOHM representation of the 3 Connection Objects	31
No. of functionally identical objects	30
No. of objects to build OCS solution (excluding previously created Function and Connection Objects in OS110 and CS110)	7
No. of Function Objects (excluding previously created Function Objects in OS110)	3
No. of Connection Objects (excluding previously created Connection Objects in CS110)	4
Re-used object count	11
Re-used object list	A10 for A11, A12, A13, A14, A16 R10 for R11, R12, R13, R14, R15, R16, R17, R18, R19, R23 E1 for D10, D14, R20, R21 L1 for D11 E2 for D12 A2 for D13, R22 A1 for D15 N1 for D16 C112 for D17 C111 for D18 N2 for D19
Eliminated object count	28
Eliminated object list	A11, A12, A13, A14, A16, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, D10, D11, D12, D13, D14, D15, D16, D17, D18, D19
No. of overall objects saved	24
No. of equivalent hypermedia objects saved	28

Table 8.4: Summary of the re-use taking place within the OCS solution of Figure 8.9.

Table 8.4 summaries the re-use taking place within the OCS solution of Figure 8.9. It shows that the application of this OCS solution leads to the generation of 7 new OCS objects comprising 3 Function Objects and 4 Connection Objects. This compares with the 31 objects that would have to be created if it were not possible to apply the OCS data model. This is calculated by adding up all the objects within the FOHM structure of Figure 8.8. Thus the OCS solution of Figure 8.9 makes an overall saving of 24 OCS objects. However, because 4 of those objects are Connection Objects (considered to be lightweight objects), there is actually a saving of 28 equivalent Function Objects.

8.6.4. Scope of the OCS Data Model

In terms of purity of principle it would be ideal if the structure author could choose, at some indeterminate point within the lower level of connections within hypermedia structure, when to stop applying the OCS data model. But is this a practical or sensible action to carry out?

The question is asked, because it is possible that the OCS data model could be applied to the internal connections within structure indefinitely. For example, the OCS data model can be applied to top-level structure. E.g. OHP-Nav hypertext links or FOHM Association relationships. This is the approach strongly recommended by this thesis to be adopted for all hypermedia structure. The OCS data model can then be applied to the Connection Objects generated as a result of having previously applied the OCS data model to top-level structure. Such an act has just been demonstrated in Subsections 8.6.2 and 8.6.3. The OCS data model can then be applied to even lower level structure, e.g. to the connections produced as a result of having previously applied the OCS data model to Connection Objects. The process could then continue even further by applying the OCS data model to the internal connections subsequently produced after the previously described application of the OCS data model. And so the practice could continue.

The question should really be at what point is it advisable to stop separating the functional and connectional properties of low-level Connection Object structures? This is because ultimately the connections between structural objects (i.e. Function Objects) must be explicitly declared at some point. If the Connection Objects are forever being broken down into more and more fine grained structure, then Function Objects will never be able to be joined together (via Connection Objects) to enable creation of appropriate high-level structures, such as hypertext or taxonomic links.

Subsection 8.6.3 already answers this question. It shows that application of the OCS data model when organising the internal connections within Connection Objects does not actually produce useful structure. This is for two reasons.

First, applying the OCS data model to Connection Objects does not enable a great deal of heavyweight object re-use to occur. This is because Connection Objects do not contain a great deal of data, i.e. they are lightweight objects as explained in Section 7.9. This stems from the objects within a Connection Object only being object stubs. The only values they contain are their identity, a marker indicating if they are instance objects, or an anchor value. Thus there is no significant heavyweight object data to re-use.

Second, application of the OCS data model to the internal connections within an OCS Connection Object does not fare better when attempting to save on the number of needlessly created objects. Without the OCS data model only the original Connection Object exists. But application of the OCS data model to a Connection Object leads to many more Function and Connection Objects being created. E.g. Figure 8.9 shows that 7 new OCS objects are created in the example scenario of Subsection 8.6.3. However as already stated these additional objects do not bring any significant re-use opportunities. The opportunity for re-use would be much improved if the object stubs were already recorded separately from the Connection Objects and contained a greater amount of data (i.e. heavyweight data). But this is not the case. Thus applying the OCS data model to Connection Objects actually creates more objects than it saves, and incurs the cost of extra object management and storage.

Therefore it can be concluded that the scope of the OCS data model should end with its application to top-level hypermedia structures such as OHP-Nav and FOHM structures. However, a different conclusion would be reached if the Connection Objects (after the first application of the OCS data model) did contain a greater amount of information, then it would be useful and sensible to continue applying the OCS data model in order to continue the process of re-organising internal structure. Perhaps in the future as more connection information is assigned within hypermedia objects, it would be beneficial to once again re-apply the OCS data model at lower level connections within internal structure.

8.7. Summary

This chapter has described how the OCS data model can be applied to the OHP-Nav and FOHM data models. A major problem with the original representation of OHP-Nav and FOHM structures is that their objects contain embedded object references.

However the OCS data model approach of dividing OHP-Nav and FOHM hyperstructures into separate Function and Connection Objects has shown that much more hyperstructure re-use is now possible. This is because it removes the hypermedia object reference embedding within hypermedia objects. Thus Computer Science Contribution 2 is fulfilled as the OCS data model promotes the general re-use of hypermedia structure.

Furthermore, this chapter has shown how the OCS data model enables the re-use of individual hypermedia objects and hypermedia structure segments which satisfies Problem Domain Issues 1 and 2. Moreover such re-use means that generally fewer objects need to be generated in order to create the same hypermedia structures using the original OHP-Nav and FOHM approaches.

Also demonstrated is how the OCS data model realizes Computer Science Contribution 1 by extending the concept of open hypermedia to individual hypermedia objects and hypermedia segments. This is because the OCS data model enables individual hypermedia objects to be attached to one another without having to alter the content of either attached hypermedia object. And the same applies to hypermedia structure segments, as structure segments can be attached to individual hypermedia objects or other structure segments without having to alter the content of either participating structures.

The chapter has also explained how the OCS data model can be used to emulate OHP-Nav's structural connections. This ability is useful for assisting those users already familiar with the connection arrangements of existing hypermedia data models. And, also examined has been the scope of the OCS data model in respect of whether it is effective to apply the OCS data model to OCS Connection Objects. However, it was determined that the adoption of the OCS data model should end with its application to top-level structures only, e.g. OHP-Nav and FOHM structures.

Chapter 9. Applications for Versioning

9.1. Introduction

This chapter explores how the Object and Connection Space data model benefits the versioning of hypermedia structure. Its main contribution is that it prevents revision proliferation. This is due to the OCS data model removing object reference embedding. A second important contribution is that the OCS data model provides a simple mechanism enabling different views of hypermedia structure versions to be presented to versioning clients. Consequently the OCS data model satisfies Computer Science Contribution 4 (Chapter 1) and resolves Problem Domain Issue 3 (Chapter 6).

CSC4:	Improved versioning of hypermedia structure.

PD3:	The dangers of revision proliferation.

9.2. Recap on Revision Proliferation

The background as well as advantages of versioning hypermedia structure have been described in Chapter 4. At the same time the problem of revision proliferation was also discussed (Section 4.6) [Conradi and Westfechtel 1998]. What follows is a brief recap.

Essentially the problem of revision proliferation occurs when attempting to create a new revision of an existing hypermedia object. Revision proliferation causes the unnecessary creation of new hypermedia objects being appended to a hypermedia structure. Section 6.4 highlighted how it is a problem for conventional organisations of hypermedia structure, such as the OHP-Nav and FOHM data models. This is due to the embedding of references between hypermedia objects within the hypermedia objects themselves. Section 6.4 reported how the OHP-Version protocol and

Contextualised Connections approach were investigated as potential solutions, but failed to prevent revision proliferation from occurring.

9.3. OCS Solution to Revision Proliferation

It is through separating the functional and connectional roles of hypermedia objects, via the OCS data model, that solves the revision proliferation problem (PD3). The result is that when a new revision of an existing Function Object is created, only the one new Function Object will result. Moreover the OCS data model ensures that the new hyperstructure formation will be exactly as expected. Also importantly the presentation of the different revisions of the hypermedia structure will be unmistakable to clients.

The scenario presented by Figure 6.9 of Section 6.4.1 can be used as an example to demonstrate the OCS solution to the revision proliferation problem. The desired *before* and *after* hyperstructure formations are shown in Figure 9.1. The intention is to create a new revision of object 'A2 v1' such that the new revision replaces the original revision within the existing structure. The new structure formation is shown as Hyperstructure B.

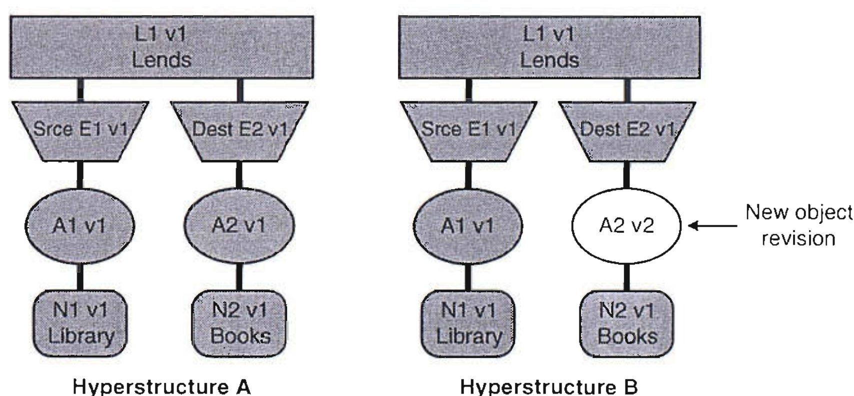


Figure 9.1: The before and after hypermedia structures[11] having created a new object revision.

Figure 9.2 shows an example of the Object and Connection Space arrangements that can be employed to record both the original and the new revision of object A2. The

[11] Neither hyperstructure's connections (between objects) are endowed with object reference embedded blocks or directionality arrows since the two hyperstructures are generic hyperstructures and are not representative as having been organised according to a specific hypermedia data model (as yet).

Object and Connection Spaces also record each hyperstructure formation (i.e. Hyperstructure A or B) as a different hyperstructure revision to ensure that the correct hyperstructure containing the appropriate revision of Anchor A2 is created. This is further explained in Section 9.4.

Connection Object C203 produces Hyperstructure A and Connection Object C204 generates Hyperstructure B.

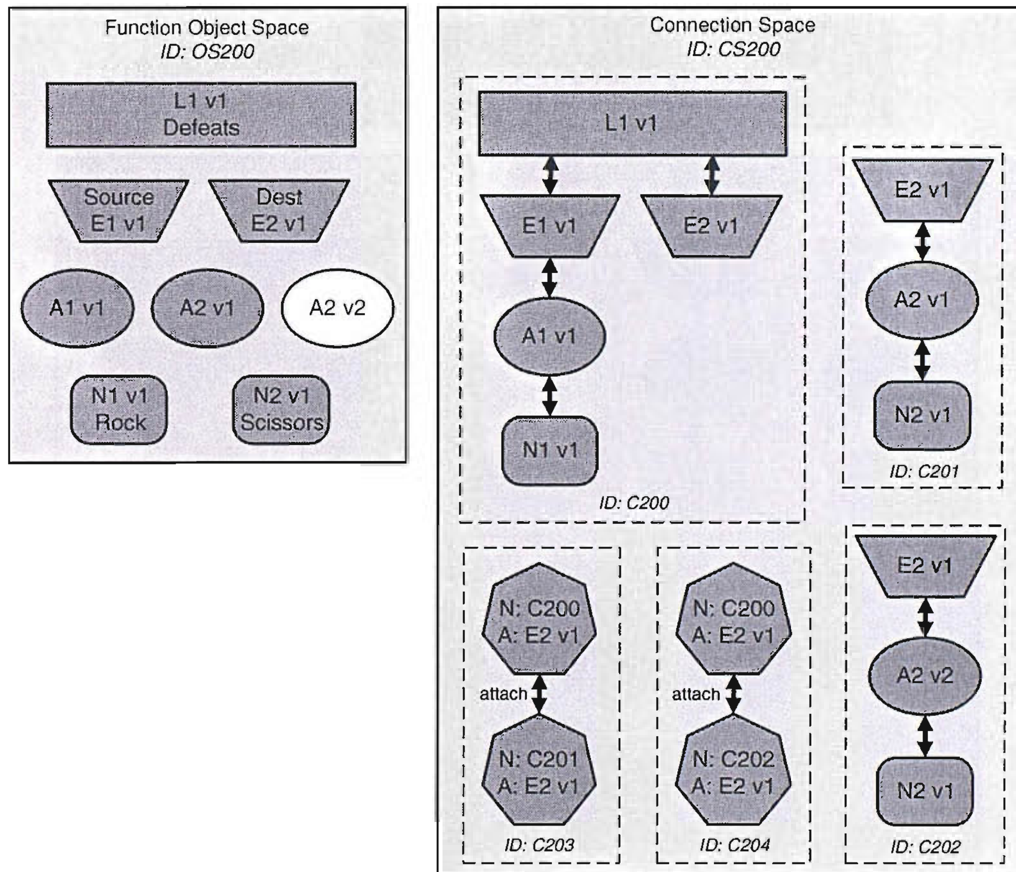


Figure 9.2: Object and Connection Spaces for Figure 9.1.

The Connection Space of Figure 9.2 is not the only Connection Space arrangement that can produce the desired hyperstructure revisions. Many different Connection Spaces can be used to describe each hyperstructure revision. Figure 9.3 provides two further examples of different Connection Spaces that are equally valid for generating the correct hyperstructure revisions. Connection Objects C210 and C211 of Connection Space CS210 produce the respective original and new hyperstructure revisions. And Connection Objects C224 and C225 of Connection Space CS220 also produce the respective initial and new revisions of hyperstructure.

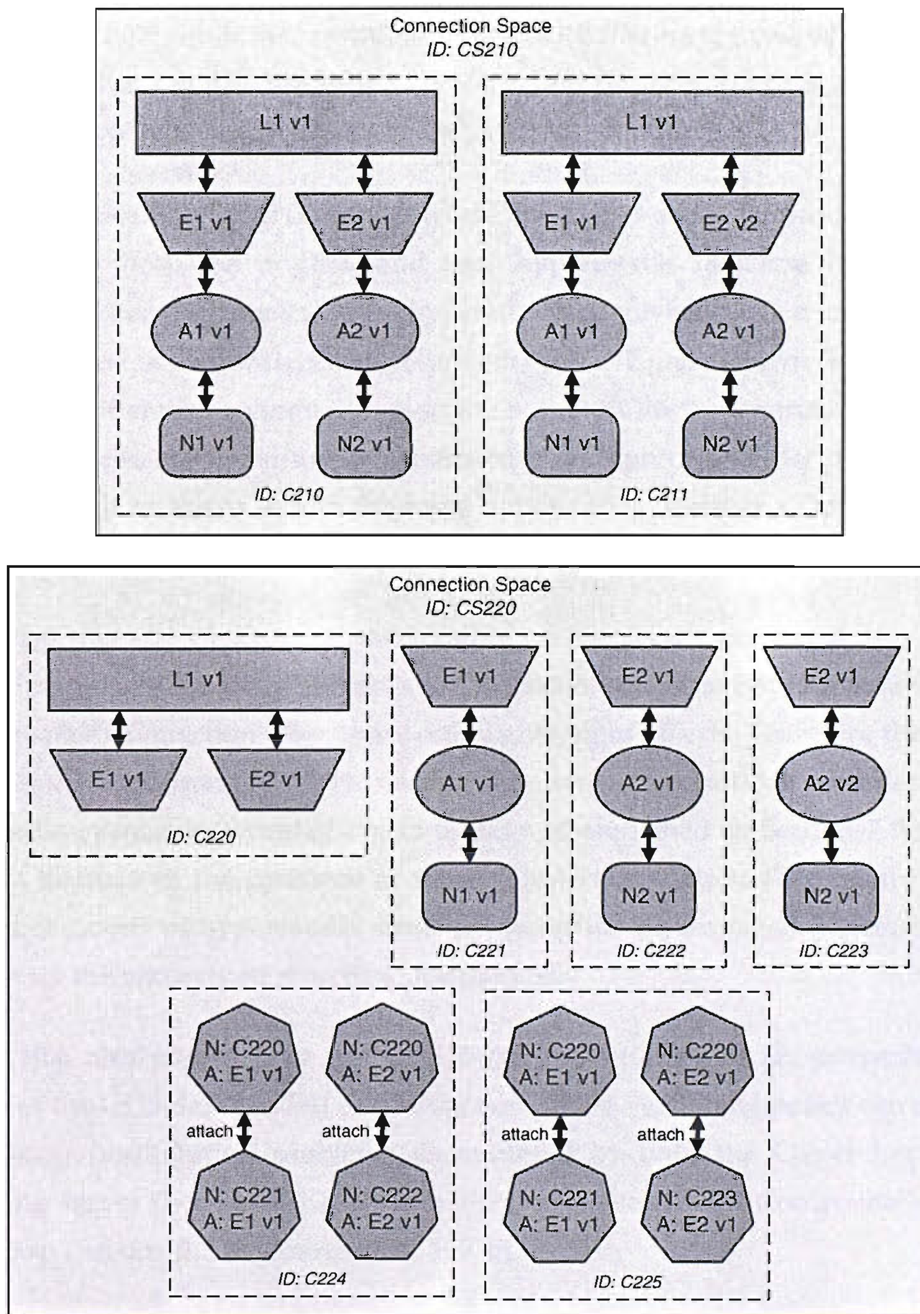


Figure 9.3: More Connection Spaces for Figure 9.1.

For all OCS solutions the Function Object Space will contain the same Function Objects as shown in the Function Object Space of Figure 9.2. These are the original 7 Function Objects used to create the initial hyperstructure revision plus an additional Function Object ('A2 v2') used to replace 'A2 v1' to create the second hyperstructure revision.

The contents of the Function Object Space demonstrate that the OCS approach generates only 1 new Function Object for each new Function Object revision created. This is unlike the conventional approach of embedding object references that can

lead to a seemingly random array of new data objects being created when only 1 new object is expected. (In the case of the OHP-Version protocol example of Section 6.4.1 two additional objects were needlessly created.)

The three Connection Space examples create the correct and expected hyperstructure formations for both the original and new hypermedia structure revisions. Each Connection Space represents each hyperstructure revision as a separate entity (Hyperstructure A separate from Hyperstructure B) as shown by Figure 9.1. Consequently there is no confusion as to the nature of the hyperstructure in terms of whether it represents the entire hyperstructure revision or whether it is acting as a container for all revisions within the same hypermedia structure. (This was noted as a potential problem within Section 6.4.1.)

However it must be conceded that overall more objects are generated as a result of the OCS approach. This is in regards to Connection Objects. But this is not really a big issue since Connection Objects are only lightweight objects. Therefore they do not contain much information which means they are not a particular burden for the hypermedia system in terms of object storage as explained in Section 7.9. But it is precisely because of the presence of these Connection Objects that ensure that the different revisions of hypermedia structure comprise the expected Function Objects and take up the anticipated structural formations.

Section 10.6 also shows how the OCS framework (which is an extension of the concept of the OCS data model) in conjunction with a versioning policy can eradicate the revision proliferation problems encountered by both the CoVer hypermedia versioning server (Section 4.6.2) [Haake 1994] and the Nested Composite Model of HyperProp (Section 4.6.3) [Soares et al. 1993b].

9.4. Different Views of Hypermedia Structure

A further advantage of the OCS data model is that Connection Spaces allow different revisions of the whole hypermedia structure to be constructed without having to embed additional versioning data within Function Objects. This capability is akin to open hypermedia's ability to present different views of networks of documents by changing the links between the documents as once again none of the connected documents need to be modified in order to participate in each hypermedia network view.

In the case of the OCS data model, it uses Connection Objects to represent the different revisions of a hypermedia structure. Thus for the example hypermedia structure revisions of Figure 9.1 the first hypermedia structure revision (represented by Connection Object C203 of Figure 9.2) is made up of Connection Objects C200 and C201, and the second hypermedia structure revision (represented by Connection Object C204) is composed of Connection Objects C200 and C202.

The versioning alternatives for representing different views of hypermedia structure revisions are often inefficient compared with the OCS approach. There are two obvious alternatives to the OCS approach:

1. The first alternative is to have one hypermedia structure that comprises both revisions within itself. Hyperstructure B of Figure 6.9 shows an example of such a hypermedia structure. It comprises both the hypermedia structures depicted in Figure 9.1. This solution requires the hypermedia structure server to have much more complicated functionality. The server must be able to distinguish that there are two or more revisions of a hyperstructure within the same hypermedia structure, and it must also be able to determine which objects belong to which revision of the structure. Moreover this approach will also have impact on normal (non-versioning) hypermedia structure navigation as the normal operation of the hypermedia structure server will have to change as well. For example clients may be forced into having to specify additional versioning operations when navigating or writing new objects to the server as each object will be treated as a revision regardless of whether it is going to be versioned or not. The end result is that clients may have to have an understanding of versioning even if it is not relevant to their work.

2. The second alternative is to create separate hypermedia structures that reflect the different revisions of the one hypermedia structure. Figure 9.4 shows an example of two separate structure revisions for the example hypermedia structures of Figure 9.1. The first revision would be composed of the original objects of the original structure, but the second revision would be composed of entirely new objects – no re-use would be possible. This solution leads to the creation of seven additional objects. Clearly this is wasteful of resources as only one object has actually changed.

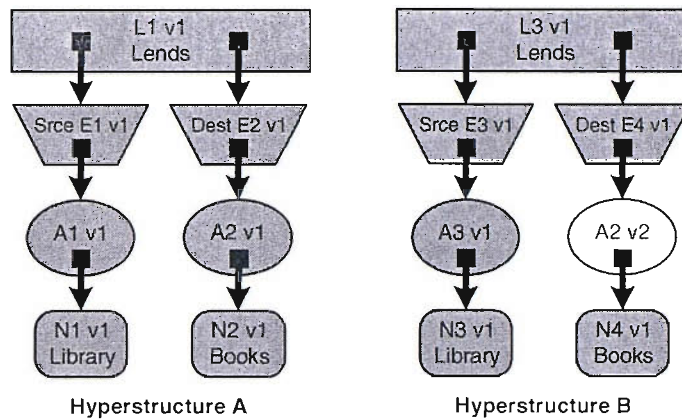


Figure 9.4: Independent structures as OHP-Nav structure revisions.

Adoption of the OCS approach means that the normal operation of the hyperstructure server does not necessarily have to change when versioned objects are introduced into the system. This is because when the OCS approach has already been used to represent the hypermedia structure, little effort has to be undertaken to organise the Connection Objects to express the hypermedia structures in their different revisions.

The ability to display different revisions of hypermedia structure also applies to the general case. That is to say that the OCS data model can display *any* views of hypermedia structure without having to embed any data within the Function Objects themselves.

9.5. OCS Data Model vs. Contextualised Connections

This chapter has already demonstrated how the OCS data model resolves the revision proliferation problems faced by the OHP-Version protocol. This has been via the example scenario of Section 9.3 which compared the OCS solution against the OHP-Version revision proliferation problems described in Section 6.4.1. Thus what remains is to identify how the OCS data model improves upon the FOHM Contextualised Connections approach.

A major failing with the Contextualised Connections approach is when attempting to append an ordinary non-versioned object to an existing FOHM structure. Section 6.4.2.4 identifies the root of the problem:

The problem is that object references continue to be embedded as part of a FOHM object's content.

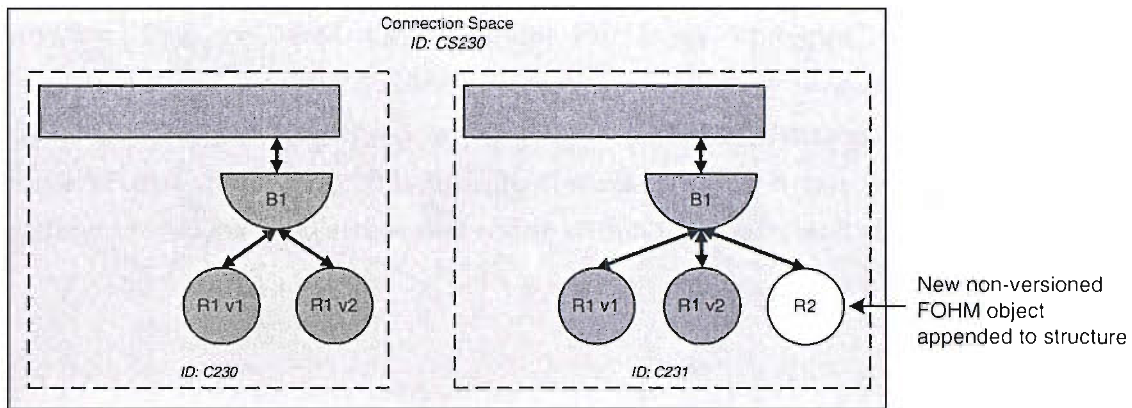
Revision proliferation results because any existing FOHM object that is to reference the new object must have its connection data (i.e. embedded object reference) updated in order to reference the new FOHM object. But this means that the existing FOHM object's content is being modified. Therefore versioning policy dictates that a new copy must be created of that updated object. Thus a new (and unnecessary) copy of that object will have to be created. The original copy preserves the original embedded connection data, whilst the new copy contains the new reference.

The OCS data model automatically resolves this Contextualised Connections revision proliferation problem because it removes object reference embedding. Thus, whenever a non-versioned object (or even a revision of an existing object) is appended to existing objects within a structure, no unnecessary objects are created. This is because there are no embedded object references that need to be updated within OCS structures.

Instead, the existing Connection Object that records the connections for the existing structure can be updated by being re-organised to include the addition of the new (non-versioned) object. Thus the only additional object required is the new object itself.

Alternatively, if the evolution of the Connection Object structure from one structural layout to another is to be preserved, then a new Connection Object can be created. The original Connection Object will record the state of the Connection Object before the new (non-versioned) object was appended, and the new Connection Object will record the updated structural layout containing the newly appended object.

Figure 9.5 shows how the connections of the FOHM Contextualised Connection example of Figure 6.14 (of Section 6.4.2.4) can be modelled using OCS Connection Objects.



(a) Original Connection Object prior to appending object R2.

(b) New Connection Object after R2 has been appended to B1. Note that Connection Object C230 continues to exist in order to record the previous FOHM connection layout.

The structural layout adopted within the figure assumes that both revisions of object R1 are to be present within the same hyperstructure revision. Also note that there is no virtual object R1. Instead B1 points directly to 'R1 v1' and 'R1 v2'.

Figure 9.5: Modelling the connections of the FOHM Contextualised Connection example of Figure 6.14 (of Section 6.4.2.4) using OCS Connection Objects.

9.6. Summary

The purpose of this chapter has been to explain how the OCS data model benefits the versioning of hypermedia structure. Consequently the chapter has also explained how the OCS data model achieves Computer Science Contribution 4.

The OCS data model's approach of removing object reference embedding from hypermedia objects means that revision proliferation can be significantly reduced if not eliminated. Thus the unnecessary duplication of hypermedia objects usually brought about as a result of structure versioning is prevented. The benefits are not only the prevention of wasteful storage of objects, but also the prevention of the formation of potentially confusing hypermedia structures. Hence Problem Domain Issue 3 of Chapter 6 has also been resolved.

Moreover the OCS data model also eases the process by which client applications can view different revisions of a hypermedia structure. This is because the OCS data model enables individual hyperstructure revisions to be preserved as separate structures by being recorded within separate Connection Objects. Thus clients only have to consult a single object (i.e. a Connection Object) in order to immediately identify which hypermedia objects are members of a given hypermedia structure

revision. This compares with alternate versioning strategies, such as the OHP-Version protocol, which combine different hyperstructure revisions within the same structure. This has the significant drawback that the hypermedia structure server must be that much more functionally complex so that it can distinguish between different revisions of structure that reside within the one overall structure.

Chapter 10. A Versioning Framework

10.1. Introduction

The principles of the OCS data model (i.e. object and connection separation) can be used as the basis for creating frameworks in order to establish relationships between any and all generic objects.

The intent of this chapter is to show how the OCS data model can be used to create relationships beyond the traditional relationships considered thus far, i.e. the joining together of hypermedia objects to form structures for the navigational, taxonomic and spatial domains. The example considered is a framework for a hypermedia versioning scheme.

The emphasis is to show how the OCS data model is more flexible and efficient compared with object reference embedding. In the process this chapter provides further evidence as to how the OCS data model addresses Computer Science Contributions 1, 2, 3, 4 and 5.

CSC1:	Extending the concept of open hypermedia into the realm of hypermedia structure.

CSC2:	Promoting the general re-use of hypermedia structure.

CSC3:	Offering a more logical approach to hypermedia structure representation.

CSC4:	Improved versioning of hypermedia structure.

CSC5:	Enabling improved hypermedia structure maintenance.

10.2. Versioning Framework

The reason why a versioning framework in particular is being investigated is because, as discussed in Chapter 4, this is an oft talked about area within the hypermedia community where there is (as yet) no ideal hypermedia versioning system.

But the aim of this chapter is not to suggest an ideal hypermedia versioning system. Its real aim is to show how the OCS data model can be used to form relationships between any generic objects. Hence this versioning framework is a theoretical exercise. However the versioning framework is still useful in providing an indication as to how the OCS data model can improve the overall versioning process (CSC4).

10.3. Versioning Framework Organisation

The role of the versioning framework is to record revision information about Function and Connection Objects. When an update is made to a Function or Connection Object, a new revision of the affected object is created and the original revision is preserved.

The versioning framework is comprised of the following 5 basic objects:

- *Function Objects.* These are the objects being versioned and are exactly the same as conventional Function Objects (Section 7.7.1).
- *Connection Objects.* Like Function Objects, these are the objects being versioned and are exactly the same as conventional Connection Objects (Section 7.7.2).
- *Revision Information Objects.* Store revision data about Function or Connection Objects, e.g. creation date and modification date.
- *Version Managers.* Group together related Function or Connection Object revisions.
- *Revision Tree Connection Objects.* Record the evolution history of Function or Connection Objects, i.e. how each revision is related to one another in terms of sibling, child or parent.

The versioning framework is represented as an underlying Object and Connection Space shown by Figure 10.1.

All framework objects of the Function Object Space are treated as generic Function Objects, and they are all connected to one another via Connection Objects (such as those in the Connection Space shown). Such an arrangement means that the relationships between framework Function Objects can be altered any time by simply re-organising the framework Connection Objects. No framework Function Objects ever need updating during such organisation since they are not embedded with any connection data. The advantages gained from this function and connectional separation are the subject of Section 10.5.

Note, that when a Connection Object is subjected to versioning, it is treated the same as a versioned Function Object. Hence the versioned Connection Object appears as a 'Function Object' within the Function Object Space in Figure 10.1.

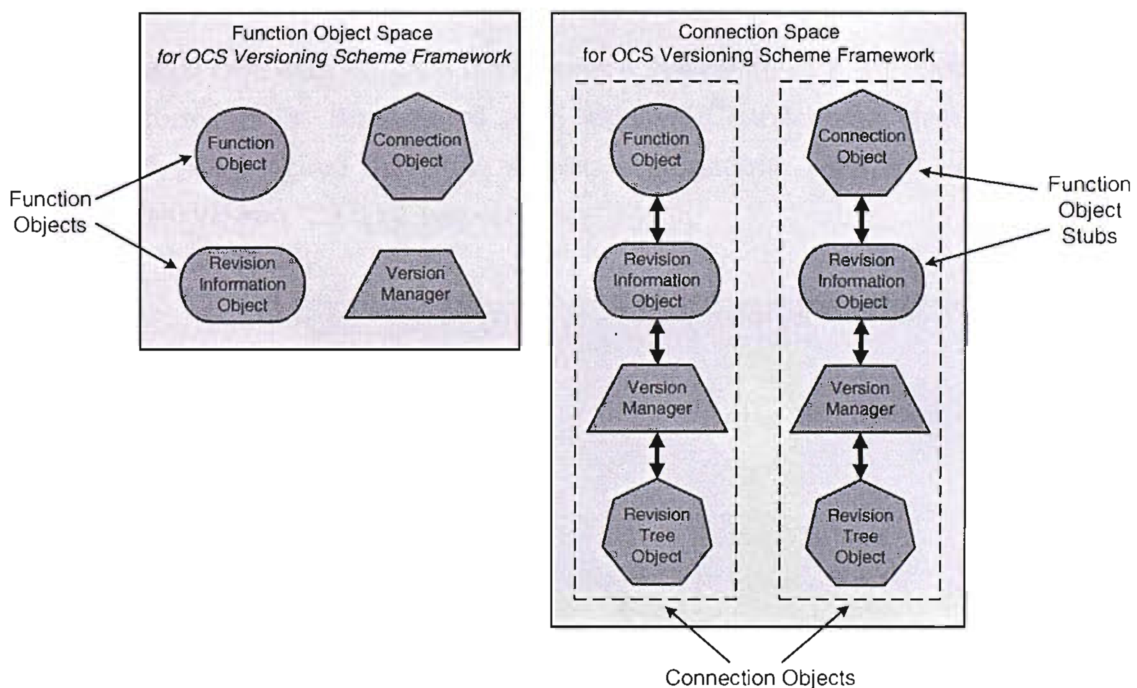


Figure 10.1: The OCS Versioning Framework.

10.4. Versioning Example

This section describes an example of a FOHM segment of structure being versioned. It centres on a structure author who intends to create a new revision of Reference Object 'R1 v1' of the hyperstructure of Figure 10.2(a) so that new revision 'R1 v2'

replaces it within the new hyperstructure formation shown by Figure 10.2(b). Every Function Object minus the Reference Function Object is re-used in both structures.

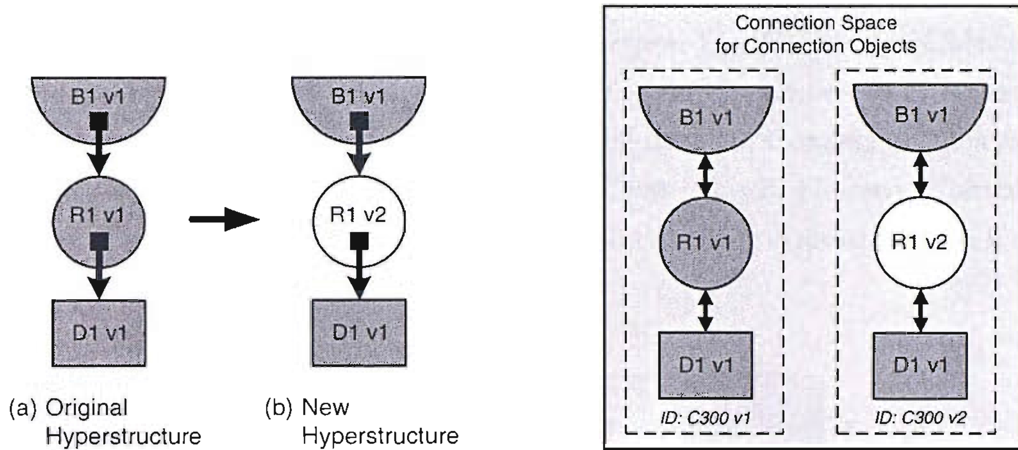


Figure 10.2: Original and new hyperstructure revisions.

Figure 10.3: Connection Objects for hyperstructure revisions.

As is the usual case with the OCS data model (Chapter 7), each different hypermedia structure formation is represented as a separate Connection Object. Figure 10.3 shows both the original and new structure formations recorded as Connection Objects 'C300 v1' and 'C300 v2' respectively.

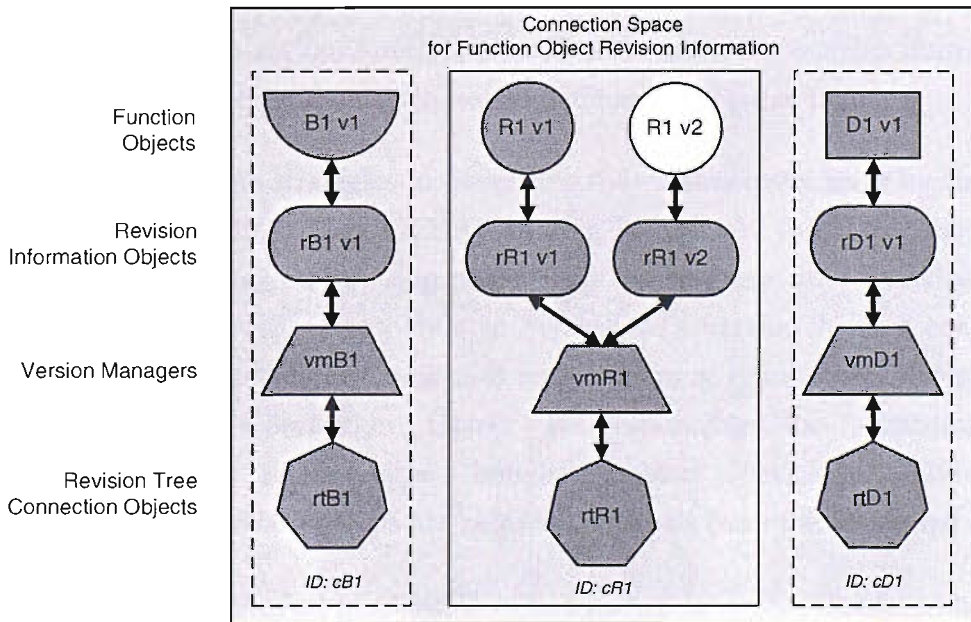


Figure 10.4: Function Object revision information.

The revision information for each Function Object is captured by Revision Function Objects and Function Object Version Managers. The revision information is related to each Function Object via Connection Objects as shown by Figure 10.4.

The evolution of each revision is captured by Revision Tree Connection Objects. The relationships between the different related revisions are shown in the Function Object Revision History Connection Space of Figure 10.5. Connection Object rtR1 shows the evolution of Reference Object 'R1 v1' to 'R1 v2'. However Connection Objects rtB1 and rtD1 only contain one member since only one revision of each object has thus far been created.

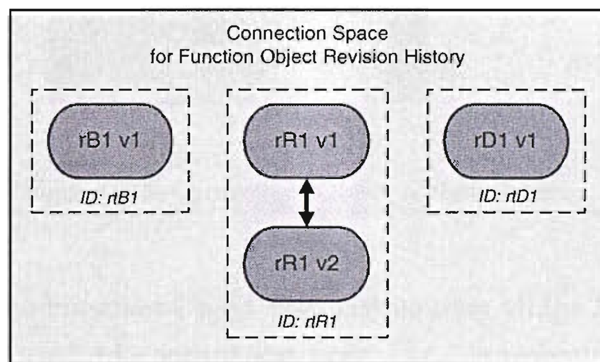


Figure 10.5: Function Object revision history.

Figure 10.6 shows the two Connection Spaces used to record the revision information and evolution history about the two Connection Objects of Figure 10.3.

In respect of Whitehead's strategies for recording the version histories of hypermedia resources [Whitehead 2001b], the OCS Versioning Framework fulfils the criteria for adopting the Versioned-Object approach. This is because the Revision Tree Connection Object (which corresponds to Whitehead's version history container) records the relationship between itself and its members as referential. Furthermore, the Revision Tree Connection Object is responsible for recording the predecessor/successor relationships between object revisions. Thus the predecessor/successor relationships are recorded separate from the versioned objects themselves.

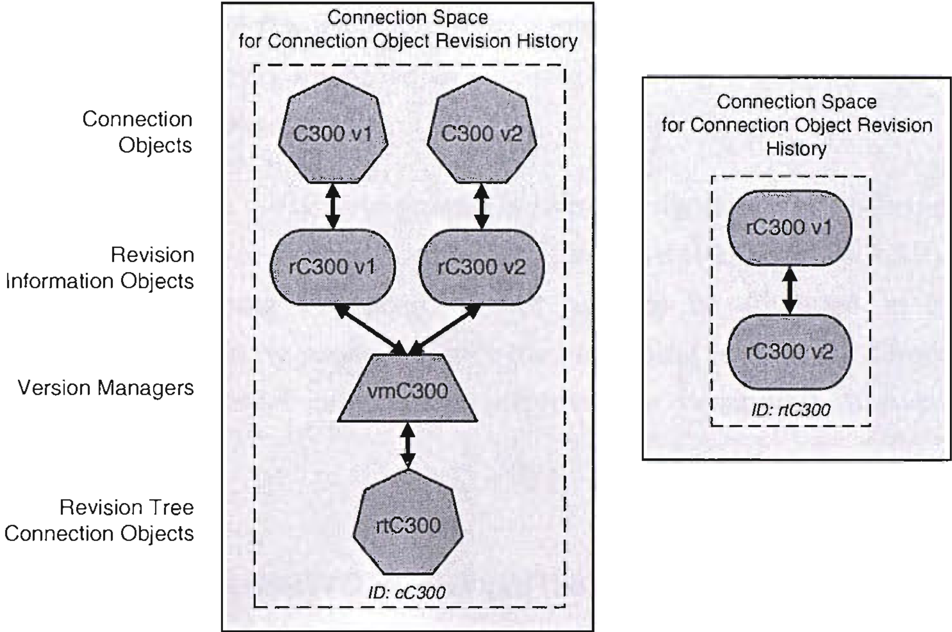


Figure 10.6: Connection Object revision history.

Figure 10.7 shows the Function Object Space containing all the Function Objects (i.e. versioning objects) used to capture the versioning information that describe the transition of the structural segment from Figure 10.2(a) to 10.2(b).

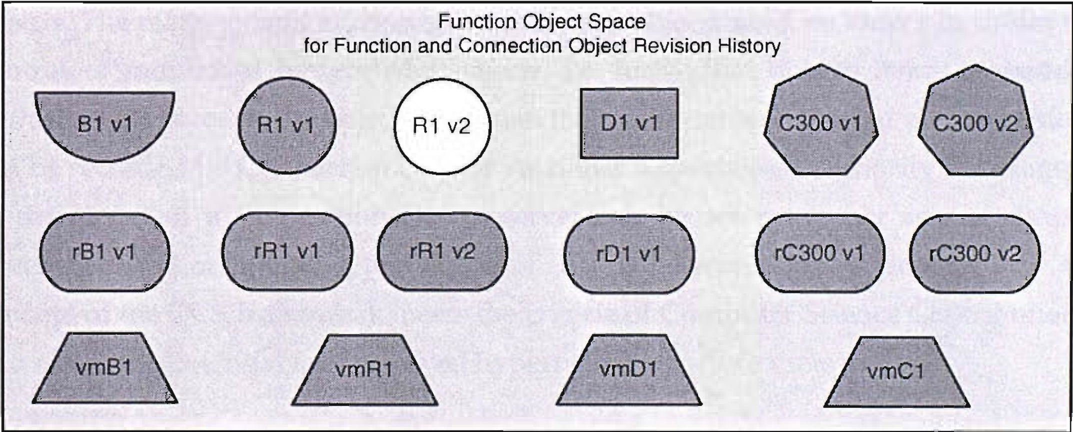


Figure 10.7: Function Object Space for Function and Connection Object Revision History.

The important point to note about the versioning framework is that all framework objects (Function Objects, Connection Objects, Revision Information Objects and Version Managers) are made to relate to one another via Connection Objects. This means that none of the original Function or Connection Objects ('B1 v1', 'R1 v1', 'R1 v2', 'D1 v1', 'C300 v1' and 'C300 v2') being versioned need to be updated with any additional versioning information as a result of them being versioned. This is

possible because the OCS framework superimposes the connection data that connects the versioning information (recorded in separate Function Objects) to the original objects now being versioned.

The net result is that the versioning process of hypermedia structure no longer needs to be as much of an "all-or-nothing" approach ([Østerbye 1992], Section 4.5.7). This is because the objects being versioned do not have to be amended in order to participate in the versioning process. Hence the versioning framework demonstrates how the OCS data model can further improve the versioning of hypermedia structure (CSC4).

10.5. OCS Framework Advantages

Applying the principles of the OCS data model for relating generic objects within a generic framework offers substantial advantages over using embedded references. Three are listed.

10.5.1. Connection Integrity

An OCS framework provides a better guarantee of connection integrity between objects. The management of connections between objects need no longer be under the control of individual hypermedia objects, i.e. individual objects being embedded with the references of the objects to which they are attached. Instead all connections can be recorded by Connection Objects such that a specialised authority for example in the shape of a Connection Object server can be set up to act as a dedicated component for maintaining just the connections between objects. In this way the concept of the OCS framework meets the criteria of Computer Science Contribution 5 as it offers the potential for improved hypermedia structure maintenance.

10.5.2. Framework Alterations

Because all objects are referenced by Connection Objects, this means that all objects can be re-positioned anywhere within the framework without needing to update the objects themselves. I.e. objects can be moved to be associated with different objects, or re-used to be associated with multiple objects simultaneously.

This is possible because such object movement does not affect the objects themselves. For example a user may decide to re-position the Version Manager object of the

versioning framework so that it is associated directly with Function and Connection Objects instead of with Revision Information Objects. The new arrangement is shown in Figure 10.8. This is a very straightforward operation as it is only necessary to modify the Connection Objects that dictate the object associations and not the objects (represented as Function Objects) themselves. This shows how the OCS framework meets Computer Science Contribution 1 as all framework objects are open (Section 7.2). That is, one framework object can be connected to another framework object without either having to be amended.

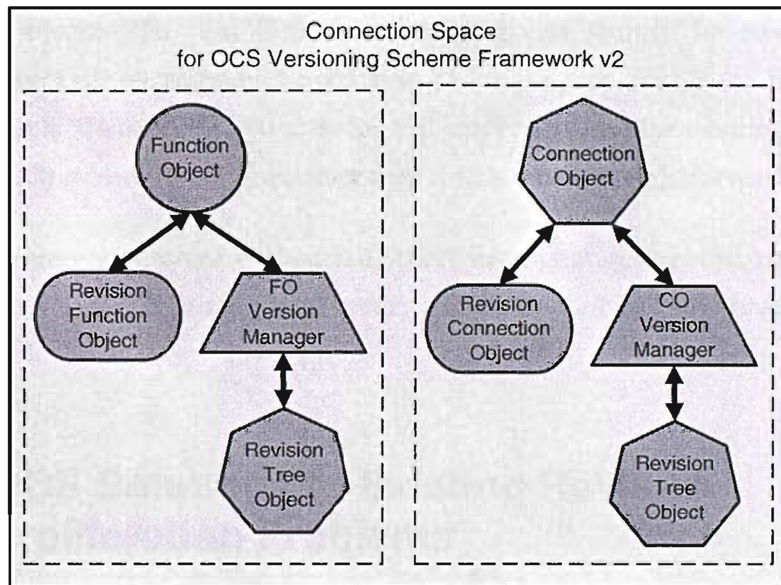


Figure 10.8: Re-organised Versioning Framework.

Furthermore individual objects can even be re-used within completely different frameworks altogether. Re-use of the framework objects within another framework has no side-effects on the other framework or vice versa. Hence Computer Science Contribution 2 is met as the OCS framework promotes the general re-use of framework objects.

Such object movement would not be possible if the object references were embedded in the objects themselves. Every time an object is moved to be associated with different objects, either or both sets of objects would have to be updated with new object referencing data. Furthermore independent re-use of individual objects would also not be possible.

10.5.3. Easier Connections

Devising Connection Objects in order to describe connections between framework Function Objects is an easier process compared with choosing within which objects to embed object references. This is because designing the connections within the Connection Object does not have as critical an impact as choosing where to embed object references.

If it is decided that the wrong connections have been established within a given Connection Object, then that Connection Object can simply be re-configured to reflect the correct connections without having to make any changes to the framework Function Objects themselves. Such a logical approach to representing hypermedia structure (CSC3) makes its maintenance that much more straightforward (CSC5).

If the object references were embedded, then such changes would not be possible without having to make changes to every embedded object involved in one such connection.

10.6. OCS Solutions to Existing Revision Proliferation Problems

The OCS data model's approach to modelling relationships within a generic framework can offer alternate solutions for the CoVer and Nested Composite Nodes revision proliferation examples of Sections 4.6.2 and 4.6.3 respectively.

10.6.1. CoVer

The CoVer versioning approach to tackle revision proliferation (Section 4.6.2) [Haake 1994] can be simplified if versioning via the OCS framework. This subsection describes one possible OCS framework solution for the CoVer revision proliferation scenario depicted by Figures 4.4 and 4.5. The scenario is to enable new node revision 'A2 v2' to replace node 'A2 v1' within the hypertext subnetwork.

The OCS solution is to use the OCS framework to model every CoVer revision object (composites, nodes and links) as an OCS framework Function Object. The connections between the objects can also be modelled as Connection Objects. Hence the example scenario presented by Figures 4.4 and 4.5 can be modelled as shown by Figure 10.9.

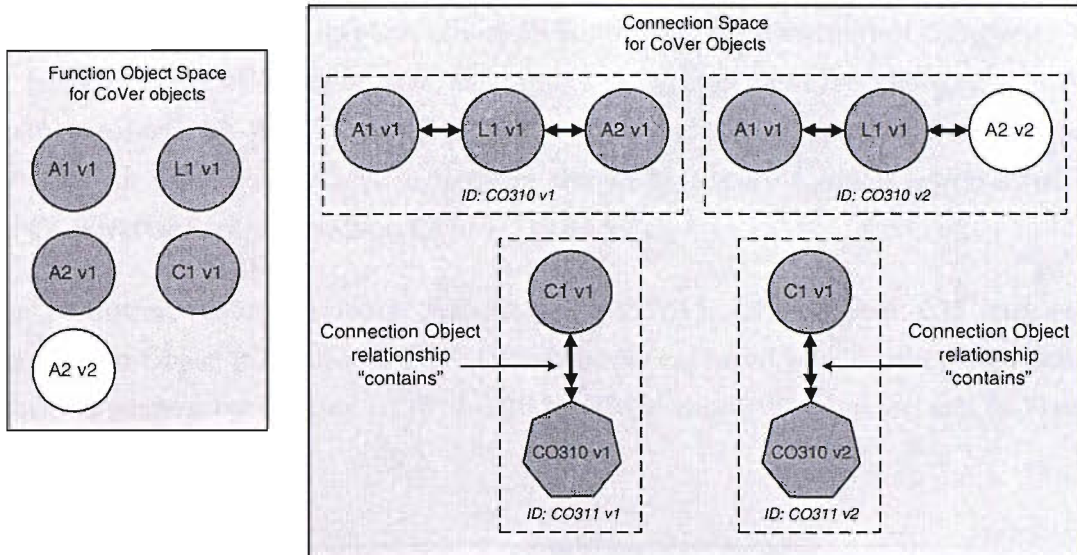


Figure 10.9: Object and Connection Spaces for the CoVer versioning example.

The OCS framework solution consists of 5 Function Objects ('A1 v1', 'L1 v1', 'A2 v1', 'A2 v2' and 'C1 v1') and 4 Connection Objects.

The objects of Figure 4.4(a) are depicted by the four Function Objects in the Function Object Space: 'A1 v1', 'L1 v1', 'A2 v1' and 'C1 v1'. The connections between the objects of Figure 4.4(a) are represented by Connection Objects 'CO310 v1' and 'CO311 v1'. 'CO310 v1' depicts the connections between nodes A1 and A2 via link L1. And Connection Object 'CO311 v1' shows the relationship between composite 'C1 v1' and the original node and link revisions (grouped together via 'CO310 v1'). Hence 'CO311 v1' portrays the original state of the overall hypertext subnetwork.

When a new revision of node object 'A2 v1' is created (via the OCS framework), only a new revision of that object is created. This is 'A2 v2' which is added to the Function Object Space of Figure 10.9. No other new Function Object revisions need to be created. This is the desired result of the CoVer example, but the CoVer versioning solution also creates the potentially confusing implicit revisions as well.

The new arrangement of the connected node and links revisions ('A1 v1' connected to 'L1 v1' connected to 'A2 v2') is captured within Connection Object 'CO310 v2' (which itself is derived from 'CO310 v1'). Neither a new explicit or implicit object revision needs to be created of link 'L1 v1' (unlike with CoVer) since only its connection arrangement has changed which is recorded by the Connection Object. The new node and link arrangement are also members of composite 'C1 v1', therefore they are connected to 'C1 v1' as depicted by Connection Object 'CO311 v2'. Once again,

neither a new explicit or implicit object revision need to be created of composite 'C1 v1' (unlike with CoVer) since only its connection arrangement has changed which is again recorded by the Connection Object. Hence the new state of the hypertext subnetwork containing 'A2 v2' otherwise shown by Figure 4.4(b) is represented by the OCS framework Connection Object 'CO311 v2'.

The evolution history of each Function Object (A1, L1, A2 and C1) and each Connection Object (CO310 and CO311) can also be captured within other Connection Spaces as shown by Figures 10.10 and 10.11. These mimic the version sets of Figure 4.5.

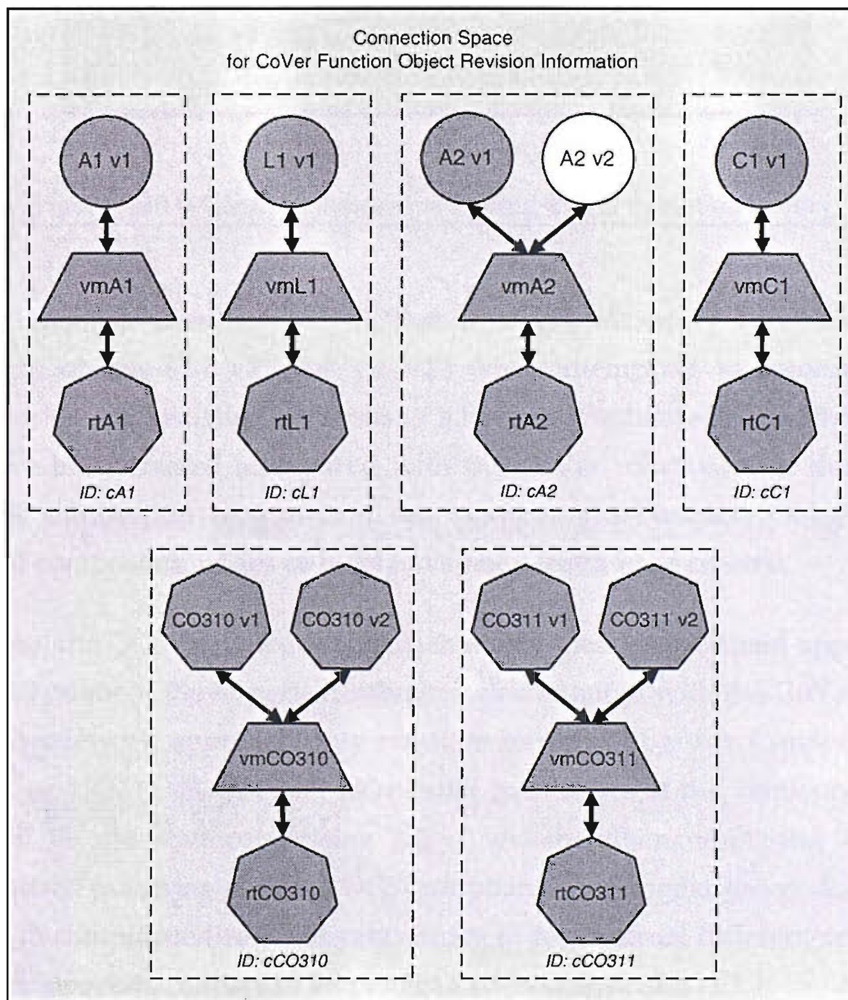


Figure 10.10: Connection Space recording CoVer revision information.

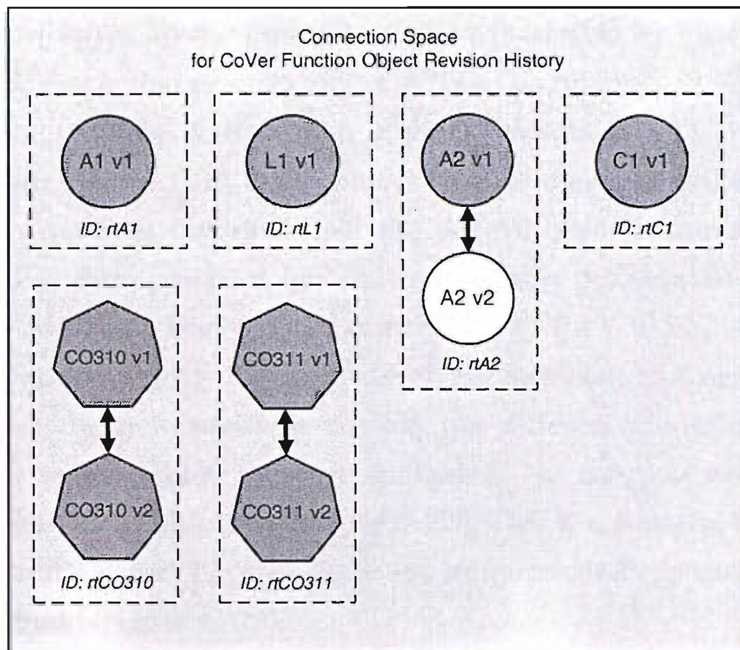


Figure 10.11: Connection Space recording CoVer evolution history.

What this example demonstrates is that it is not necessary to create duplicate heavyweight objects ('L1 v2' and 'C1 v2') when attempting to associate a newly created revision with existing revisions of a hypertext subnetwork. Admittedly more objects have been created (compared with the CoVer solution), but these are only lightweight Connection Objects and not heavyweight Function Objects, i.e. new revisions of composites, nodes or links have not needed to be created.

Furthermore, the OCS framework approach offers a less complicated approach to re-creating each state of the hypertext subnetwork compared with the CoVer approach. The OCS framework approach only requires retrieval of either Connection Object 'CO311 v1' or 'CO311 v2' in order to re-build both states of the versioned hypertext subnetwork, i.e. one state comprising 'A2 v1' and the other comprising 'A2 v2'. This benefits pattern matching since an OCS-compliant hypermedia server does not need to partake in complicated re-build procedures to re-construct different revisions of a hypermedia structure that are to be pattern matched against.

10.6.2. The Nested Composite Model

The OCS framework can provide an alternate solution for the NCM Propagation Guided By Perspective approach (Section 4.6.3) [Soares et al. 1993b].

This subsection focuses on the example scenario presented by Figures 4.6 and 4.7. The desired solution is that new revision 'D v2' can be referentially contained by the same User Context Nodes that contain original revision 'D v1'. This is shown in Figure 10.12. But, because each User Context Node contains an embedded reference to the children nodes it contains, then the desired scenario cannot be achieved without the mass duplication of objects (i.e. revision proliferation) as shown by Figure 4.6(b). Here, many heavyweight objects ('C v1', 'E v1', 'B v1', 'A v1' and 'F v1') are being duplicated where the only difference between the new and original revisions is that the new revisions contain the different identifiers of the User Context Nodes within which they are contained. No function data of any User Context Node has changed (other than in 'D v2'). This is a wasteful solution. Soares at al. concur, and as such have devised the compromise Propagation Guided By Perspective solution (Figure 4.7(b)).

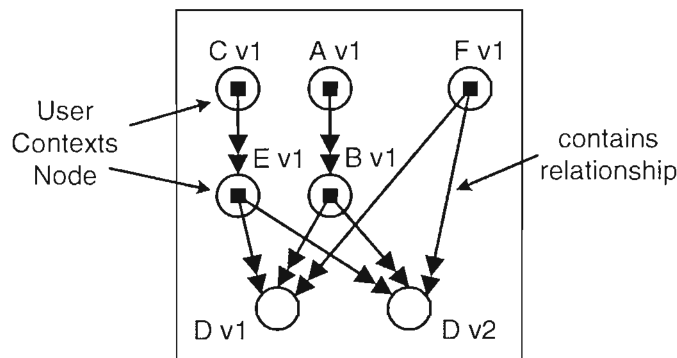


Figure 10.12: Desired NCM solution for HyperProp revision proliferation problem.

However, the OCS framework can produce the desired solution without causing the revision proliferation of heavyweight objects. This is achieved by the OCS framework modelling every NCM User Context Node as a Function Object; and modelling the relationship of one User Context Node being contained in another User Context Node via Connection Objects. Hence the initial and desired scenarios presented by Figures 4.6(a) and 10.12 respectively can be depicted by the Object and Connection Spaces of Figure 10.13.

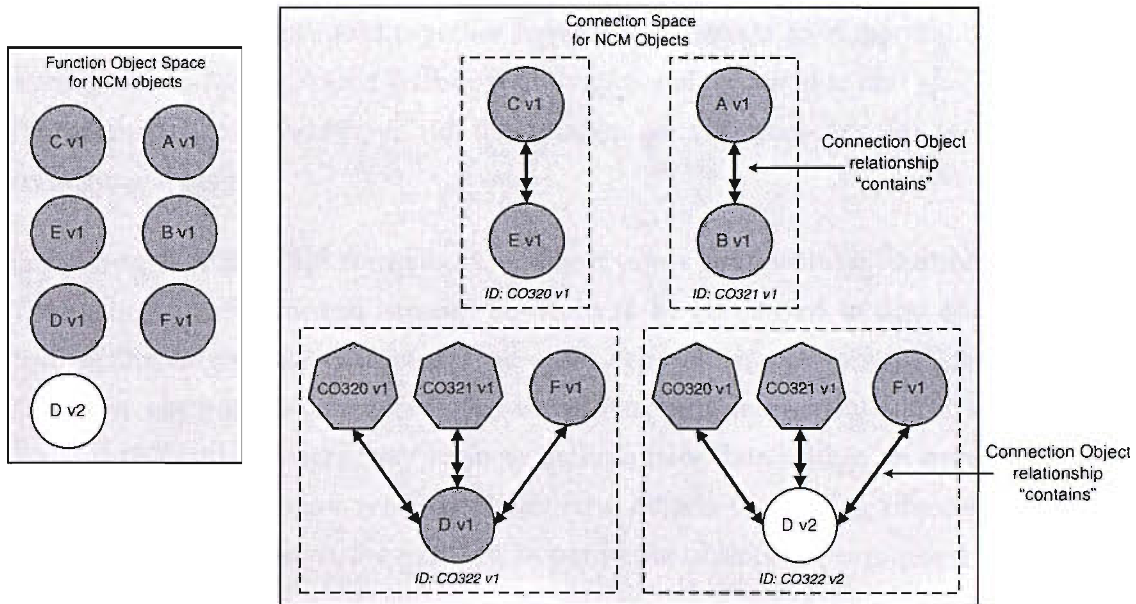


Figure 10.13: Object and Connection Spaces for NCM example.

The hypertext subnetwork of Figure 4.6(a) is modelled by the Function Object Space containing the 6 revisions, 'C v1', 'E v1', 'D v1', 'A v1', 'B v1' and 'F v1', along with Connection Objects 'CO320 v1', 'CO321 v1' and 'CO322 v1' in the Connection Space. The initial hypertext subnetwork arrangement is represented by 'CO322 v1'. It shows 'D v1' contained by 'E v1', 'B v1' and 'F v1'; 'E v1' contained by 'C v1', and 'B v1' contained by 'A v1'.

The hypertext subnetwork of Figure 10.12 can easily be created by the OCS framework – all that needs to be done is the re-organisation of the connections within Connection Object 'CO322 v1'. This produces 'CO322 v2' which ensures that 'D v2' is contained within the same User Context Nodes as 'D v1'. Hence Connection Objects 'CO322 v1' and 'CO322 v2' together represent the desired hypertext subnetwork of Figure 10.12 as 'D v1' and 'D v2' are contained within the same User Context Nodes. All this has been achieved without any revision proliferation of User Context Nodes.

The evolution history of the User Context Nodes (which would be captured by Connection Objects) is not shown as hopefully this should be intuitive.

10.7. Summary

This chapter has shown how the OCS data model can be used to provide a framework for governing the relationships between any and all objects within a hypermedia environment. That is to say the OCS data model can be used in a greater

capacity than to simply join together hypermedia objects to create the conventional linking structures employed within the navigational, taxonomic and spatial domains. The example considered within the chapter is a framework for a hypermedia versioning scheme.

The strength of the OCS framework is that it offers the flexibility that objects (of the framework) can be moved around at whim to be connected to any and all objects (within the originating framework or other separate frameworks). This is a useful capability since connections between objects are not necessarily static. For example hypermedia system users may wish to include new data within an existing system, to achieve this they may wish to attach new objects to existing objects in order to contain this new data. If the existing hypermedia objects are organised as Function Objects within Connection Objects, then this can be more easily achieved compared with if the hypermedia objects contained object reference embedding data.

This chapter has also provided further examples as to how the OCS data model meets Computer Science Contributions 1, 2, 3, 4 and 5.

- *CSC1*: The OCS framework has extended the concept of open hypermedia (Section 7.2) into the realm of generic hypermedia structure so that all generic objects are open.

- *CSC2*: The OCS framework also promotes the general re-use of hypermedia structure since framework objects cannot only be re-used within their original framework, but within completely independent frameworks too.

- *CSC3*: The OCS framework also offers a more logical approach to hypermedia structure representation as the connections between framework objects are not inhibited by object reference embedding.

- *CSC4*: Although not the specific aim of this chapter, the OCS versioning framework has provided further evidence as to how the OCS data model can improve the versioning of hypermedia structure.

- *CSC5*: The OCS framework has also showed how the principle of object and connection separation (the cornerstone to the OCS data model) can benefit hypermedia structure maintenance.

Chapter 11. Applications for Link Maintenance

11.1. Introduction

This chapter explores how the Object and Connection Space data model can improve hypertext link maintenance. The chapter focus is on repairing the specific structure of hypertext links because, as identified in Chapter 5, this is a commonly recurring problem quoted in hypermedia research literature. But the solutions offered in this chapter can just as equally be applied to the maintenance of all types of hypermedia structure.

In the process this chapter explains how the OCS data model satisfies Computer Science Contributions 3 and 5 (of Chapter 1) and resolves Problem Domain Issue 4 (of Chapter 6).

CSC3:	Offering a more logical approach to hypermedia structure representation.

CSC5:	Enabling improved hypermedia structure maintenance.

PD4:	The confusion caused by repairing broken internal routes.

11.2. Benefiting Link Maintenance

This chapter focuses on three primary ways that the OCS data model can benefit hypertext link maintenance (CSC5):

1. The breakdown of OCS hypermedia objects into Function and Connection Objects provides a more logical approach towards repairing broken hypertext links.

2. The ease with which hypertext links can be versioned as a result of the OCS data model (as demonstrated by Chapters 9 and 10) also offers an opportunity to improve link maintenance.

3. Link maintenance can be further improved due to the connections between Function Objects being recorded separately within Connection Objects. This means that the focus of connection maintenance can be concentrated on just the connections between objects.

11.3. A More Logical Approach

Section 6.5 of the Problem Domain Chapter explained how object reference embedding muddies the process of repairing broken internal routes within hypermedia structure. This can be remedied by the OCS data model as it prevents a seemingly random selection of objects being updated when repairing broken hypertext links. This is what tends to be the case when repairing hypertext links whose hypermedia objects are connected together via object reference embedding as explained in Section 6.5.3.

The broken hypertext link scenario of Section 6.5.2 depicted by Figures 6.15 and 6.16 can be used as an example. Figures 11.1 and 11.2 show a possible OCS representation for the corresponding broken and repaired hypertext links.

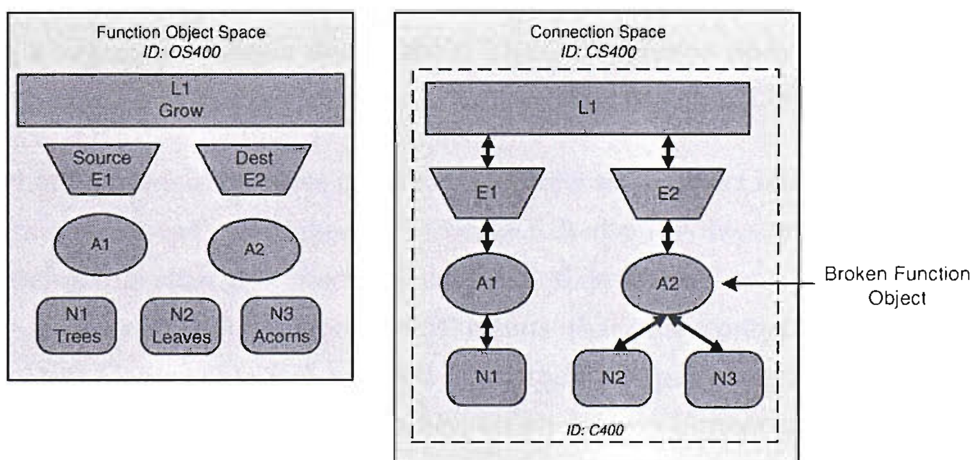


Figure 11.1: Before repair – the OCS representation of hypertext link of Figure 6.15.

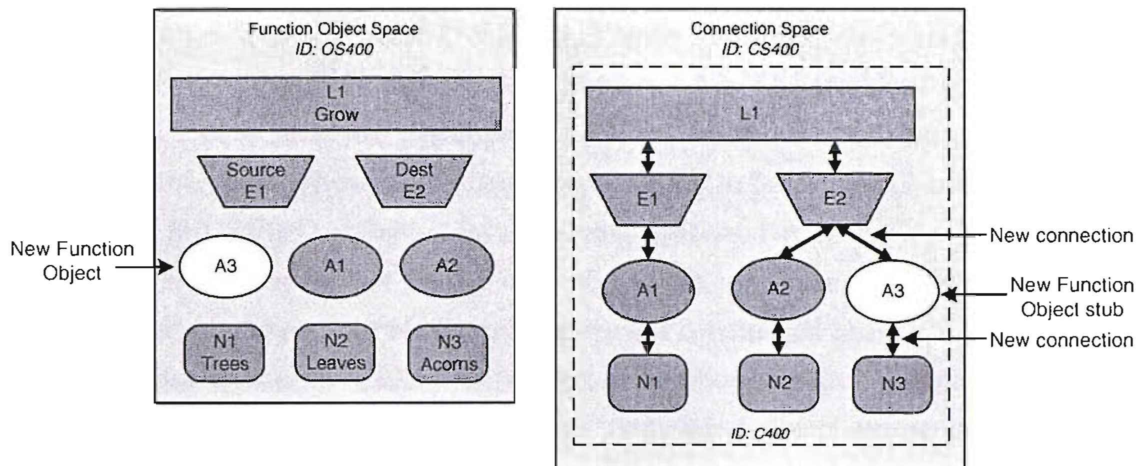


Figure 11.2: After repair – the OCS representation of hypertext link of Figure 6.16.

The main benefit of the OCS data model is that it restricts the repair of broken object functionality to be carried out within the Function Object Space, and it restricts connection re-organisation to only be enacted within the Connection Space. This makes it clearer to understand which objects are to be updated and why.

For example, in the case of the broken hypertext link of Figure 11.1, Anchor A2 is identified as the broken object. The solution (as prescribed by Section 6.5.2) is to create a new copy of the Anchor object and update it with the correct anchor data for the modified 'Acorns' document. Anchor A2 cannot itself be updated since it is required for the 'Trees Grow Leaves' route. The OCS solution is to repair this broken functionality within the Function Object Space. It does this by creating the aforementioned Anchor object (shown as Function Anchor Object A3 in Figure 11.2) and adding it to Function Object Space OS400. The OCS solution does not update any other Function Objects since the functionality of no other Function Objects is broken.

To ensure that both internal routes point to the correct node object resources, the link repair solution (as already described in Section 6.5.2) also involves re-organising the overall hypermedia structure. Because the OCS data model restricts connection updates to Connection Objects only, this means that this connection update is restricted to just Connection Space CS400. This update is performed by inserting a new Function Object stub that represents new object A3 into Connection Object C400. Stub A3 is then connected to existing stubs E2 and N3, and stub N3 is disconnected from stub A2. No Function Objects are updated as part of the object re-organisation process since no object functionality of any Function Objects has been updated. This is because the premise of the OCS data model is that no Function Objects store connection data.

This is a much clearer method compared with the object reference embedding approach to link repair as described in Section 6.5.3. The problem with the latter approach is that it does not distinguish between repair to object functionality and repair to connection organisation. Thus, as described in Section 6.5.3, object reference embedding link repair can be a very confusing process. For example when repairing the broken hypertext link of Figure 6.15, the object reference embedding approach actually updates the content of objects that are not considered broken, e.g. E2 and A2. The reason for their update is not to fix broken functionality, but to repair the overall structure of the link by re-directing their embedded object references. But such distinctions are not made clear by the object reference embedding approach. Hence uncertainty can result as to why particular objects (and not other objects) have been updated. This problem is not encountered within the OCS data model since connection updates are restricted to Connection Objects. Therefore the OCS data model does not confusingly appear to update unbroken hypermedia objects. Thus the repair of internal routes within hypermedia structure is much more comprehensible. In this way the OCS data model satisfies Computer Science Contribution 3 and Problem Domain Issue 4 as well as enabling overall improved hypermedia structure maintenance (Computer Science Contribution 5).

11.4. Versioned Hypertext Links

Another way that the OCS data model can improve link maintenance is through versioning the hypertext links themselves. As has been seen in Chapter 9 the OCS data model offers a straightforward approach for carrying out hypermedia structure versioning.

However, the fact that hypertext links are independently stored and managed from the documents they reference opens up further risks for broken hypertext links [Davis 1999; Ashman 2000a]. This can take the form of deliberate or accidental hypertext link manipulation where the individual objects that make up a hypertext link may be deleted or their contents changed. The net result being that the hypertext link is now broken.

The introduction of versioned hypertext links offers the means to combat these potential problems. This is by freezing hypertext links prior to any changes being enacted upon them. Thus if broken, the hypertext link can be rolled back to its previous working state.

A greater advantage is gained when hypertext link versioning is used in conjunction with an independent document versioning scheme. If old hypertext link arrangements are preserved then they can be used to reference previous document revisions. This makes traversal between previous document revisions possible as versioning clients can follow the previous revisions of the hypertext links that existed between those old document revisions. Moreover if the user decides to return the documents back to their previous state via the document versioning scheme, then their associated hypertext links will also be able to be restored as well.

11.5. Finding Connections Between Objects

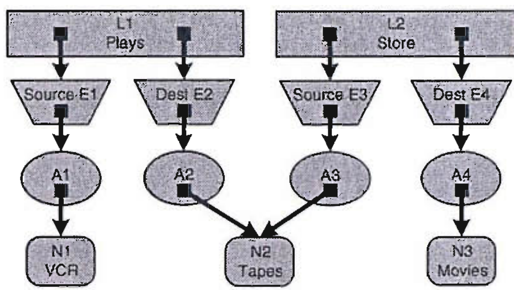
Another positive outcome of the OCS data model is that the functional and connectional aspects of hypermedia objects (i.e. Function and Connection Objects) can be managed by separate components. This provides a good opportunity for improved hyperlink maintenance as link maintenance can be focused specifically on, for example, a specialised Connection Object server as described in Section 10.5.1. This offers two notable advantages:

1. Maintenance is more straightforward. Connection Objects are lightweight objects. This makes them easier to deal with since they only describe connection information. Therefore it is quicker and easier to examine and monitor them as there is less data to wade through to in order to carry out link maintenance. This is contrasted with conventional hypermedia objects composed of embedded object references where there is more data to analyse since they store the function data in addition to the connection data.

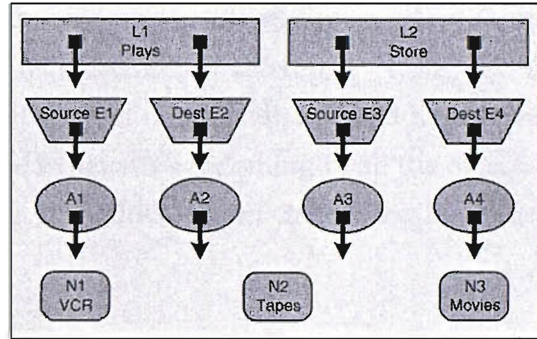
2. The OCS model allows the option of using a single Connection Object to record the Function Objects that comprise an entire hypertext link. This means clients do not have to spend time and effort querying the entire Storage Back End database searching for each connected object. Thus when Function Objects have to be updated or Connection Objects re-arranged as part of the link repair process, it is easier to locate the Function Objects in order to carry out that repair.

   This is in contrast with the conventional object reference embedding approach. Here, exhaustive searching must always be carried out to determine which objects are connected to one another. There is always a degree of uncertainty for the client as to whether they have located all hypermedia objects that are

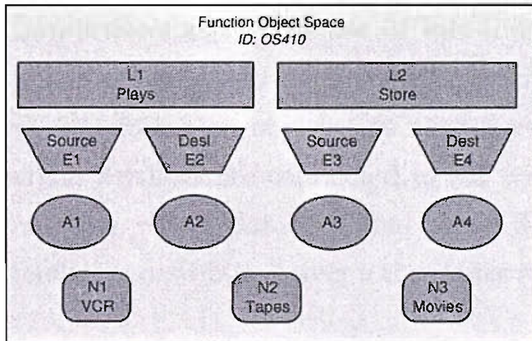
members of a given hypertext link. This adds doubt as to whether all relevant objects have actually been repaired.



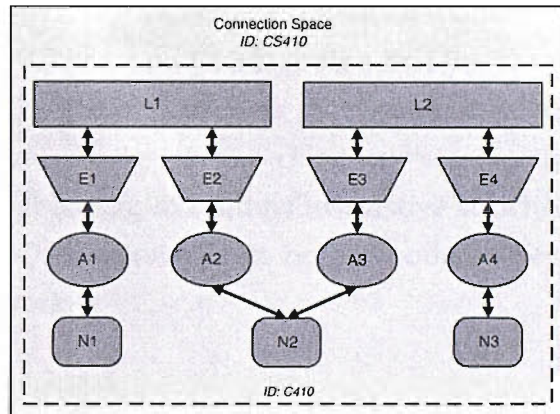
(a) OHP-Nav representation of Hyperlink A.



(b) Storage of the individual OHP-Nav objects of Hyperlink A when recorded within a Storage Back End database using embedded object references.



(c) OCS Function Object Space for Hyperlink A.



(d) OCS Connection Space for Hyperlink A.

Figure 11.3: Locating connections within OHP-Nav versus the OCS data model.

Figure 11.3 enables a comparison to be made between how the hypermedia object members of a hypertext link can be located when a hypertext link is represented using the OHP-Nav object reference embedding data model and the OCS data model.

The hypertext link in question is shown in Figure 11.3(a). It is recorded as Hyperlink A. Figure 11.3(b) shows how the individual objects of OHP-Nav Hyperlink A would be recorded within a Storage Back End database when adopting the conventional object reference embedding approach between objects.

Potential problems can develop when attempting to search for OHP-Nav objects that are connected together in such a linked-list embedded object reference arrangement (as described in Section 3.2.5). For example when searching for objects connected to

Node N2, after the first referencing Anchor A2 has been retrieved, how does the client know whether there are additional Anchors that reference Node N2? The solution is to perform exhaustive searching in order to be sure. In this case it will discover additional Anchor A3. But then how does the client know when to stop searching? There may indeed be more Anchors that reference Node N2 not necessarily managed by this hypermedia server, but in other distributed hypermedia servers. The client must then apply the same exhaustive searching to all the objects of the hypertext link to be certain that all hyperlink object members have been retrieved.

Figures 11.3(c) and (d) show one possible representation as to how Hyperlink A may be represented using the OCS data model. Figure 11.3(c) shows the Function Object Space and Figure 11.3(d) shows the Connection Space. Figure 11.3(d) shows the connections between the Function Objects of Hyperlink A to be contained within one Connection Object. The use of this solitary Connection Object means that the client will immediately know which Function Objects are members of Hyperlink A. As regards to which objects are associated with Node N2, the client can see exactly which Anchors are connected to N2 without having to conduct exhaustive searches over the object database that store this Connection Object or over other object databases distributed over a computer network.

11.6. Summary

This chapter has described how the OCS data model can benefit the maintenance of hypertext links. The chapter also explains how the OCS data model satisfies both Computer Science Contributions 3 and 5, and resolves Problem Domain Issue 4.

Three areas where link maintenance is significantly improved by the OCS data model have been discussed, thus satisfying Computer Science Contribution 5.

The first was the OCS data model easing the process of repairing the broken internal routes within structure without adversely affecting other unbroken internal routes. This is possible because the OCS data model provides a more logical approach to hypermedia structure representation. This addresses Problem Domain Issue 4 and meets Computer Science Contribution 3.

The second area was the ease with which hypertext links can be versioned as a result of the OCS data model. As explained this offers yet another opportunity for improving link maintenance.

The final area under discussion was the structure maintenance advantage gained by recording Function and Connection Objects separately. This setup allows the focus of connection maintenance to be more precisely concentrated on just the connections between objects enabling an opportunity for more efficient link repair to take place. Also discussed was the idea of using a single Connection Object to record the Function Object layout of an entire hypertext link. This saves on searching effort to locate Function Objects to repair and/or locate Connection Objects that may need re-organising as part of the structure repair process.

Chapter 12. Conclusions

12.1. Introduction

This chapter forms the conclusion to the thesis. A major part of the chapter considers how the OCS data model has achieved the Computer Science Contributions set out in Chapter 1 and met the Problem Domain Issues of Chapter 6.

The Computer Science Contributions restated are:

CSC1:	Extending the concept of open hypermedia into the realm of hypermedia structure.

CSC2:	Promoting the general re-use of hypermedia structure.

CSC3:	Offering a more logical approach to hypermedia structure representation.

CSC4:	Improved versioning of hypermedia structure.

CSC5:	Enabling improved hypermedia structure maintenance.

The Problem Domain Issues restated are:

PD1:	The use and storage of hypermedia objects that carry out the same function.

PD2:	The use and storage of identical hypermedia structure.

PD3:	The dangers of revision proliferation.

PD4:	The confusion caused by repairing broken internal routes.

The remainder of the chapter provides an overall view of the OCS data model, considers the OCS data model in relationship to existing research and looks at the future work that can be carried forward as a result of this research.

12.2. Restatement of the Problem

The nature of the problem is the internal organisation of hypermedia structure. As noted in Chapter 1, the role of hypermedia structure data models (e.g. OHP-Nav and FOHM) is to specify which hypermedia objects should be connected to one another. This is with a view to creating linking structures used within the navigation, taxonomic and spatial hypermedia domains.

However these same hypermedia data models also implicitly describe how hypermedia objects should be attached to one another. And it is here where the problem lies.

The typical method is to embed hypermedia object references (describing to which objects an object is attached) within the hypermedia objects themselves (Section 3.2.5). This means that these references (i.e. connectional data) become part of the content of the hypermedia object itself. As reported by the Problem Domain Issues Chapter 6 this is both a wasteful and inefficient mechanism for connecting objects.

12.3. The OCS Data Model

The Object and Connection Space data model reforms the way that the underlying structure of hypermedia data models is organised. I.e. it addresses *how* hypermedia objects should be attached to one another.

As with the majority of Open Hypermedia Systems the OCS data model maintains hypermedia objects as first class entities. However, a significant difference is that the OCS data model also acknowledges the importance of connections between hypermedia objects. Therefore it also elevates the connections between hypermedia objects to be first class too.

To this end the OCS data model separates the functional and connectional roles of hypermedia objects (Section 7.2). This is where the strength of the model lies.

- The functional role of a hypermedia object is the purpose or function that that object is programmed to carry out. For example the role of an OHP-Nav Anchor is to point at internal content within a node resource.
- The connectional role describes to which object a given hypermedia object is connected.

The OCS data model allocates the functional aspect of hypermedia objects to Function Object Spaces. They contain Function Objects which are hypermedia objects minus their connection data. The connectional roles of hypermedia objects are allocated to Connection Spaces. They contain Connection Objects which describe the connection data used for attaching hypermedia objects together.

A further characteristic of the OCS data model is that its approach to the internal organisation of hypermedia structure is analogous to the open hypermedia principle of separating links from nodes. This was remarked upon in Section 7.7.3. For example Function Objects (Section 7.7.1) are akin to OHS nodes, because, like OHS nodes, Function Objects do not contain embedded linking data. And binary Connection Objects (Section 7.7.2.1) can also be considered akin to OHS links, because, like the majority of OHS links, binary Connection Objects are used to form connections between two resources only.

However, n-ary Connection Objects (Section 7.7.2.2) are different to conventional separated OHS hypertext links. This is because n-ary Connection Objects are more like hypertext networks since they can be used to group together a set of connections that do not necessarily reference the same Function Object. The advantage offered by n-ary Connection Objects is that they enable efficient re-use for both Function and Connection Objects. This is because they allow creation of specific connection arrangements that are more suited to being re-used in the future (Section 7.7.2.3).

12.4. Scope of the OCS Data Model

The scope of the OCS data model was also commented upon in Section 8.6. Of interest was that the OCS data model was found to be inefficient when applied to the internal connections within a Connection Object. This is despite that Connection Objects are themselves a form of hypermedia structure.

The reason for this is because Connection Objects are comprised of object stubs (not independent objects) that only contain lightweight object data. Hence application of

the OCS data model to Connection Objects actually creates more objects than it saves. This results in extra object management and storage, but at the same time it does not lead to significant heavyweight object data re-use. Therefore it was concluded that the scope of the OCS data model should end at the top-level of hypermedia structure, e.g. OHP-Nav and FOHM structures.

12.5. Problem Domain Issues Answered

This section accumulates the evidence presented throughout the thesis as to how the OCS data model resolves the 4 Problem Domain Issues of Chapter 6. This section essentially forms a summary of the thesis.

12.5.1 Issue 1: Repetitive Hypermedia Objects

Sections 7.17 and 8.7 describe how the OCS data model has addressed this issue.

The concern stated in Section 6.2 is that many hypermedia objects replicate identical functionality within the same or multiple hypermedia structures. To counteract such unnecessary duplication of objects, the OCS data model facilitates individual hypermedia object re-use.

The conventional approach of hypermedia object representation prevents individual object re-use, as hypermedia objects are embedded with object reference data. If a hypermedia object were to be re-used, then many of the objects that are also members of the same hypermedia structure are forced into being re-used as well (Section 8.2).

Re-use is achieved within the OCS data model by separating the functional and connectional roles of hypermedia objects (Section 7.7). They are split into Function and Connection Objects. Hypermedia objects effectively become Function Objects, i.e. they contain just the functional purpose of a hypermedia object. They do not contain the connection data that describes which objects are connected together. Hence when a Function Object is re-used, just the single object is re-used. No other associated Function Objects are forced to be re-used, because there is no connection data present in the Function Object.

Re-use is enabled by Connection Objects (see Section 7.7.2) and Instance Objects (Section 7.11.2). The latter enables unique identification of re-used Function Objects

within the same hypermedia structure. Function Objects can also be re-used by being attached to entire structures, i.e. Connection Objects (Section 7.11.1), as well as different Function Objects.

Detailed examples of Function Object re-use are shown in Chapter 8. OHP-Nav and FOHM structures are re-organised according to the OCS data model in order to compare the OCS data model against the conventional object reference embedding approach of hypermedia structure organisation. The chapter also shows how the number of objects carrying out the same function can be reduced versus the embedded reference approach (Section 8.4).

Further examples of Function Object re-use are shown in Chapter 9. This is because versioning via the OCS data model also re-uses Function Objects within versioned structures as part of the versioning strategy.

Finally, Chapter 10 introduces the idea of using the OCS data model as a generic framework for representing all connections for all generic hypermedia structures. Framework objects are assigned as Function Objects that can be re-used anywhere within a single framework or within multiple frameworks.

12.5.2. Issue 2: Repetitive Hypermedia Structure

Sections 7.17 and 8.7 describe how this issue is met by the OCS data model.

The focus of Section 6.3 is that the segments of one hypermedia structure are often repeated within the same or other independent structures. The OCS data model employs Connection Objects to facilitate re-use of structural segments.

Conventional hypermedia structure representation makes the re-use of a solitary segment of structure largely unattainable. This is due to object reference embedding once again. Any attempt to re-use a structural segment, means that any objects outside that segment but still attached to the segment (through one of the segment objects holding an embedded reference pointing to an outside object) would also have to be re-used (see Section 6.3 and Figure 6.4).

To remedy this, OCS Connection Objects group together objects (Function and/or Connection Objects) describing how they are connected to one another. They enable re-use by not writing any connection data within the connected objects themselves. Hence both Connection and Function Objects can be re-used without any of their associated objects being re-used too.

Connection Objects can be organised into binary or n-ary arrangements (Section 7.7.2). This allows efficient hyperstructure re-usability as it enables small or large segments of structure to be re-used. Examples of hypermedia structure re-use via Connection Objects are shown in the "Object and Connection Space Example" of Section 7.14.

Chapter 8 shows how the conventional organisation of structure within OHP-Nav and FOHM, i.e. object reference embedding, restricts the opportunity for hypermedia connection (i.e. structure) re-use. It also illustrates how application of the OCS data model enables a block of Function and Connection Objects to be re-used without having to amend any of the objects or connections re-used within that block (Figures 8.2 and 8.3).

Examples of Connection Object re-use are shown in Chapter 9, as versioning via the OCS data model uses Connection Objects to record different revisions of hypermedia structures. Additional examples are shown in Chapter 10 where all the connections between framework objects are represented as Connection Objects. This enables the framework to re-use any and all framework objects anywhere within a single framework or within multiple frameworks.

12.5.3. Issue 3: Revision Proliferation

Section 9.6 describes how this issue is addressed by the OCS data model.

Revision proliferation is the creation of unexpected new hypermedia objects when attempting to create a single new revision of an existing hypermedia object. Once again this problem is caused by the conventional representation of hypermedia structure with embedded object references.

Chapter 9 explains how the OCS data model, through separating the functional and connectional roles of hypermedia objects and allocating them to Function and Connection Objects, solves the revision proliferation problem. This is because when a new revision of an existing Function Object is created, only the one new Function Object will be created.

The OCS data model also ensures that the new hyperstructure formation will be exactly as expected, i.e. no new unexpected revisions being added to the structure caused by revision proliferation (Section 9.3).

12.5.4. Issue 4: Improved Hypermedia Structure Maintenance

Section 11.6 describes how this issue is addressed by the OCS data model.

The focus of this problem domain issue is repairing broken internal routes within hypermedia structure without adversely affecting other unbroken internal routes within the same hypermedia structure (Section 6.5). The presence of embedded object references makes structure repair a particularly confusing process. This is because users often end up having to update the embedded references in unbroken objects in order to connect new objects that have been created as part of the new repaired organisation of structure (Section 6.5.3).

Chapter 11 demonstrates how the breakdown of hypermedia objects into OCS Function and Connection Objects provides a clearer and more logical approach towards repairing broken internal routes within hypermedia structure. This is because the repair as regards fixing functionality (e.g. changing an anchor's hotspot) is confined to Function Objects, and the repair as regards fixing the formation of structure (e.g. which object should be connected to which) is confined to Connection Objects. The net result is that only the expected objects are updated.

12.6. Computer Science Contributions

This section identifies how the OCS data model achieves the 5 Computer Science Contributions set out within Chapter 1 of the thesis.

12.6.1. CSC1: Extending the Concept of Open Hypermedia

A major contribution of the OCS data model is that it opens up hypermedia structure (Section 7.17).

Section 7.2 explains how conventional hypermedia structure is not open. This is due to the embedding of object references within hypermedia objects which amalgamates the functional and connectional roles of hypermedia structure within individual hypermedia objects.

Section 7.2 provides a definition for open hypermedia structure:

> Hypermedia structure can be said to be open if it can link to and from other structure without having to alter the (linked) structure itself.

The OCS data model satisfies this open hypermedia structure definition by separating the functional and connectional roles of hypermedia objects, i.e. removing the embedding of object references within the objects themselves.

The key to opening up hypermedia structure is the role of Connection Objects. This is because a Connection Object is able to connect Function or Connection Objects together without having to amend the content of any objects it is connecting. The result is that Function Objects can connect to other Function Objects without having to amend either Function Object, and Connection Objects can be connected to other Connection Objects without either Connection Object being modified. This is possible because Connection Objects superimpose connection data, that describes which object is connected to which, on top of the objects themselves.

Opening hypermedia structure enables opportunities for re-using structure (CSC2), improved versioning (CSC4) and structure maintenance (CSC5).

12.6.2. CSC2: Promoting the general re-use of hypermedia structure

Sections 7.17, 8.7 and 10.7 summarise how the OCS data model promotes the general re-use of hypermedia structure.

This Computer Science Contribution is achieved via the separation of the object and connection data which allows opportunity for hypermedia structure to be re-used to form new relationships between different node resources within hypermedia networks.

The overall impact is a saving on unnecessary duplication of resources (Section 7.14). Evidence of hypermedia structure re-use is provided by Chapter 8 where OHP-Nav and FOHM hypermedia structures re-use Function and Connection Objects within multiple hypermedia structures. Re-use of specific Function Object types also allows identification of other related node resources (Section 7.15.1).

Chapter 9 shows how objects are re-used to efficiently model different revisions of hypermedia structures. This is followed by Chapter 10 which shows how the OCS

framework can promote the general re-use of hypermedia structure. This is possible as framework objects cannot only be re-used within their original framework, but within completely independent frameworks too.

12.6.3. CSC3: Logical approach to hyperstructure representation

The OCS data model offers a more sensible approach for representing hypermedia structure (Section 11.6). This is due to the functional and connectional separation of hypermedia structure.

It is more logical that function information should be allocated to one object type (Function Objects), and connection information should be assigned to a different object type (Connection Objects). This is contrasted with the conventional representation of hypermedia structure that combines both types of information within a single object type, i.e. a hypermedia object with object reference embedding.

Adoption of the OCS data model results in hypermedia structure creation becoming a two step process. The functionality of an object is determined by manipulating its content; and an object's connection with other objects is determined by assigning it to one or more Connection Objects.

The benefit of this logical approach is noticeable in Chapter 11 (which is also summarised in Section 12.6.5) where the OCS data model makes the repair of broken internal routes within hypermedia structure a more straightforward process. This is because fixing the broken functionality of a hypermedia structure is confined to updating individual Function Objects, and repairing the arrangement of a structure is limited to updating Connection Objects.

12.6.4. CSC4: Improved versioning of hypermedia structure

Chapters 9 and 10 demonstrate how the OCS data model improves hypermedia structure versioning.

The OCS data model approach to versioning removes the problem of revision proliferation and ensures that the formation of the versioned structures take shape as expected. This is explained in Section 9.3.

The OCS data model also improves the overall versioning process by making clearer the presentation of the different revisions of versioned hypermedia structure to

clients. This is achieved by Connection Objects preserving the before and after versioning states of hypermedia structures. Figures 9.2 and 9.3 of Section 9.3 exemplify this feat by showing how the different revisions of overall structure are captured (within separate Connection Objects) when a revision of a Function Object is modified.

Chapter 10 also shows how the principles of the OCS data model can be used to create a versioning framework within which hypermedia structure can be versioned. The framework is able to capture revision information about individual Function and Connection Objects as well as record the evolution history of Function and Connection Objects.

12.6.5. CSC5: Improved hypermedia structure maintenance

Chapter 11 describes how the OCS data model benefits the overall maintenance of hypermedia structure.

Firstly the OCS data model clarifies the process by which broken internal routes within hypermedia structure are repaired. This has already been summarised in Section 12.6.3 and is more fully explained in Section 11.3.

Because the OCS data model makes versioning easier (Chapter 9), this also has a positive knock-on effect for link maintenance. If any changes are committed on hypermedia structure (either Function or Connection Objects) that break the structure (e.g. a Function Object within a Connection Object being deleted), then the (now) broken hypermedia structure can be rolled back to a working version (Section 11.4).

A third improvement is when entire structures are housed within single Connection Objects. This makes it easier to locate any broken Function and/or Connection Objects that need fixing since all object identifiers are made available to the repair client. This compares against the embedded object references approach where exhaustive searching is necessary since the repair client can never be certain that all hypermedia objects have been identified for repair. Section 11.5 provides an example.

12.7. Relationship to Existing Research

This section looks at a range of related research to the work on the OCS data model.

12.7.1. Xanadu

Re-use plays a key role in both Xanadu and the OCS data model. A comparison between Xanadu content lists and OCS Connection Objects is worth drawing attention to.

As explained in Section 2.7.3 Xanadu [Nelson 1999a] uses content lists as a container for loading documents with content data. This is achieved through content lists holding referential pointers that point to the addresses where document content is permanently stored. Moreover a content list can also point to another content list. This allows the content of the referenced content list to be re-used by the referencing content list in the structured order already set out by the referenced content list. The relevance is that Xanadu does not embed the data referenced by Xanadu content lists with actual connection data. It is upon this foundation that the same Xanadu content data can be re-used as members of different documents.

OCS Connection Objects are similar to content lists since both act as containers of information. In the case of Connection Objects, they contain a record of connected Function and/or Connection Objects. Like Xanadu content lists, Connection Objects also do not embed connection data within the Function and/or Connection Objects they are connecting together. Also like Xanadu, OCS Connection Objects can reference other Connection Objects so that the content of the referenced Connection Object can be re-used by the referencing Connection Object in the same structured order already set out by the referenced Connection Object. In this way both Function and Connection Objects can be re-used by multiple Connection Objects to become members of different hypermedia structures.

OCS Instance Objects can also be compared with the manner by which Xanadu re-uses the same piece of text within content lists. This is because in order to re-use the same piece of text, Xanadu creates new pointers to point at the permanent address of the text to be re-used. Thus Xanadu can re-use any original text in more than one place within a document. This is akin to how OCS Instance Objects (Section 7.11.2) enable the re-use of Function Objects within the OCS data model. This is because an Instance Object acts as a reference pointer to the actual Function Object that it represents within a given hypermedia structure.

As explained in Section 2.7.3.3 Xanadu transclusion links can be make clickable in order to identify where the same document content simultaneously exists (i.e. is being re-used) within different documents. The OCS data model can also imitate this process so it too can make shared structure visible to the user. A key feature of the

OCS data model is to enable the sharing of the same Function and Connection Objects between different hypermedia structures. This means that hypermedia structures can be queried in order to identify which OCS objects are being simultaneously re-used within other structures. Therefore if desired by users, connections can be forged between the simultaneously shared structure to highlight their re-use akin to Xanadu transclusion links.

12.7.2. The World Wide Web

As reported in Section 2.10 the World Wide Web is a closed hypermedia system as it embeds hypertext links within documents and does not store links separate from document content. This means that it is not possible to apply the OCS data model directly to Web hypertext links. Indeed the World Wide Web is a prime example of how embedding connection data can be harmful since documents containing embedded references cannot be re-used without being forced to re-use the other referenced documents.

However, certain similarities can be drawn between the OCS data model and the Semantic Web (also described in Section 2.10). For example the Semantic Web is intended as a more efficient way of representing data on the WWW. Likewise, the OCS data model also offers a more efficient way to organise structure within Open Hypermedia and Structural Computing Systems.

Of interest is also the Semantic Web's attention on improving resource discovery. This is because by different mechanisms the OCS data model can further improve resource discovery too. Section 7.15.1 describes how this is possible through the analysis of shared resource structure. For example when the OCS data model is applied to the OHP-Nav protocol, new relationships can be revealed between the node resources that hypermedia structure is being used to connect. E.g. if the same OHP-Nav Anchor is being shared between different hypermedia structures then this will identify other hypertext links that reference the same node content within any node resource. This can be useful for locating node resources that share the same or similar content.

12.7.3. Open Hypermedia

The OCS data model has a natural relationship with the concept of open hypermedia since they are both motivated by the removal of embedded connection information. For open hypermedia (Section 2.12), this is the removal of embedded hypertext link

data within document content, and for the OCS data model, this is the removal of embedded object referencing connection information within the individual hypermedia objects that make up a hypertext link. Indeed, it was this idea of dis-embedding hypertext link information from document content that inspired innovation of the OCS data model. The reasoning behind this was that if embedded connection data can be removed from the documents being linked together, then why should embedded connection data continue to exist within the hypermedia structures being used to link those same documents together?

The OCS data model also shares a direct relationship with open hypermedia since, as described in Chapter 1, the role of the OCS data model is to "conceptually re-structure the internal organisation of [open] hypermedia structure". Moreover it was the OHP-Nav protocol (an example of a standardised linking protocol) that was used to demonstrate (principally in Chapters 7 and 8) the viability of the OCS data model. The relevance here is that linking protocols are an essential element of an Open Hypermedia System since the premise of open hypermedia is to separate hypertext links from document content, and linking protocols are used by OHSs to manage the separated hypertext links. Therefore any OHS that store its links as separate entities should be able to apply the OCS data model to structurally re-organise its hypertext links. It makes no difference whether the links are compliant with the standardised OHP-Nav protocol and data model, or are proprietary to a specific OHS. Of importance is that the Open Hypermedia System represents a hypertext link as a set of individual hypermedia object types akin to the individual object types that make up OHP-Nav hypertext links, e.g. Link, Anchor and Node. Therefore the OCS data model can be applied to generic Open Hypermedia Systems in much the same way as described for OHP-Nav. I.e. separating the functional and connectional roles of individual OHS hypermedia objects into Function and Connection Objects. Examples of OHSs with proprietary linking protocols include Microcosm [Carr et al. 1994], Chimera [Anderson et al. 1994] and HyperDisco [Wiil and Leggett 1996].

The improvements will be the same as those described for OHP-Nav compliant OHSs. The first, for example will be the removal of any embedded reference connection data within individual hypermedia objects. Thus extending the concept of open hypermedia into the structural organisation of hypertext links themselves. Further improvements include enabling generic OHSs to benefit from efficient hypermedia object re-use, enhanced hypermedia object and hypertext link versioning, and improved hypermedia object and hypertext link maintenance.

Another significant relationship shared between the OCS data model and open hypermedia is the importance both attach to the concept of re-use. Open hypermedia focuses on the re-use of whole hypertext links. This is made possible through hypertext links being stored independently of document content. Such re-use enables hypertext links to point to multiple source or destination documents. Microcosm's generic links provide a good example. The OCS data model adopts the same view that re-use is important, but its focus is at a lower level. Namely the re-use of individual hypermedia objects (via Function Objects) and blocks of structure (via Connection Objects). Taken together it is these individual hypermedia objects that comprise the re-usable hypertext links of open hypermedia.

12.7.4. Structural Computing

Automatically it can be inferred that there is a natural relationship between Structural Computing and the OCS data model since Structural Computing classifies hypermedia (the branch of structure that the OCS data model organises) to be a subset of the field of Structural Computing. Therefore it follows that if the OCS data model can be applied to hypermedia then there must be some scope for applying the OCS data model within Structural Computing.

As discussed in Section 2.15 Structural Computing promotes the view that 'relationship' should be the atomic building block within a computing environment [Nürnberg et al. 1997]. To this end the 'relationship' attribute is encapsulated within the structural atom - the most fundamental basic unit of structure for building everything within the computing environment. This includes data items (e.g. a computer file) and high-level structural abstractions, e.g. hypertext links, taxonomic links and spatial associations.

The OCS data model shares this same sentiment of structure as it too utilises structure as an entity for relating items together. For example the OCS Connection Object is used to connect (i.e. relate) Function Objects and/or Connection Objects together. Moreover, both Structural Computing and the OCS data model acknowledge the critical role that structure plays at establishing relationships between elements as they both elevate structure to be a first class entity (Section 7.7).

Like the OCS data model, Structural Computing holds the view that structure should not be embedded within data. Structural Computing's stance is that structural abstractions, e.g. hypertext links and taxonomic links, should be stored separately from the data they are used to organise.

However, the difference with the OCS data model is that most Structural Computing systems continue to embed connection data within the structural atoms used to build structural abstractions. HOSS [Nürnberg et al. 1996] and Construct [Wiil et al. 2000; Wiil 2001] are examples of Structural Computing systems that embed connection data within their atomic building blocks. Therefore there would appear to be a role for the OCS data model within the Structural Computing environment where the OCS data model can be used to separate the embedded structural connection information from structural data within structural atoms. (Or stated more clearly, remove the connection information embedded within individual structural atoms.) The revised structural organisation would result in structural connections being grouped within OCS Connection Objects and the structural functional data (of structural atoms) being represented as OCS Function Objects.

The benefits for Structural Computing would be the same as for open hypermedia, namely atomic structure re-use, improved structure versioning and improved structure maintenance.

Evidence of the suitability of the OCS data model within a Structural Computing setting has been demonstrated in Chapter 8 by its application to the embedded connections within FOHM structure. The relevance of FOHM is that it provides generalised structural abstractions for the navigational, taxonomic and spatial domains.

12.7.5. Object Prototyping

The approach to re-use by Connection Objects (i.e. the ability of Connection Objects to re-use both Function and Connection Objects) enables comparisons with JavaScript object prototyping [Flannagan 1997].

As is the case with most object-oriented languages, a JavaScript class acts as a template which describes the data and behaviour associated with the objects of that class. Objects are typically created using a class constructor where all objects are initialised in the same way and each object is assigned its own copy of the class properties. However JavaScript also offers object prototyping as an alternative method for specifying the properties of an object of a class. A prototype object is a special object associated with the constructor of that class. Any properties defined by the class prototype object are shared by all objects of the class. This means objects do not get their own unique copy of the prototype properties. It is here where the comparison with the OCS data model lies.

Re-used Function and Connection Objects share similarities with object prototypes since both are examples of objects/properties that can be re-used in multiple places. Connection Objects can also be compared with JavaScript class objects. This is because, like JavaScript object prototypes, Connection Objects generally contain both shared and non-shared data. The shared data takes the form of Function Objects which can be simultaneously shared between multiple Connection Objects. The non-shared data is the record of which Function Objects are connected together.

However a central difference between the OCS data model and object prototyping is on their approaches to re-use. In the OCS data model, it is the Connection Object (i.e. an object) that orchestrates re-use. Whilst in object prototyping, re-use is orchestrated by the class itself.

Another difference is the degree of re-use each offers. Connection Objects are more flexible since individual Connection Objects can implement different re-use situations, i.e. each Connection Object can include or omit different Function or Connection Objects for re-use, and they can organise them in different linking patterns. This is in contrast with object prototyping. The only data that can be shared is that data assigned to the object prototype (whether relevant or not). Individual class objects have no choice on what is or is not shared. Furthermore individual class objects have no choice on whether to participate in the sharing process - it is always the case that all objects instantiated from the class will re-use the object prototype data.

12.7.6. Standard CB-OHS Storage Interface

The developers of Construct (described in Section 2.15.1) have been working on the definition of a standardised CB-OHS storage interface [Wiil 2000b]. This is to enable interoperability at the Open Hypermedia Framework Layer of the CoReArc data model (Figure 3.1). The Standard CB-OHS Storage Interface comprises a core set of services and a set of extensions dealing with more advanced services in the areas of access control, concurrency control, version control and notification control. The Interface has thus far been implemented as a basic service in the Construct development environment [Wiil 2001].

The OCS data model is related to the Standard Storage Interface as they both centre round the further definition of structure. In the case of the OCS data model this is the breakdown of hypermedia objects into Function and Connection Objects. The Standard Storage Interface, on the other hand, breaks structure down into Units [Wiil

2000a]. A Unit is a basic storage entity that is opaque to the system that manages and stores it. The Unit is an example implementation of Structural Computing's structural atom (Sections 2.15 and 12.7.4). It is the responsibility of the Storage Interface to provide persistent storage of Units.

A Unit is comprised of the following characteristics:

- *Binary Attribute*. Can contain binary data.
- *Attribute*. Can contain all types of attribute values.
- *Relation*. Can contain references to other Units.
- *Behaviour*. Can contain computations.

A Unit can be compared to a Function Object as they both represent hypermedia objects. But the major difference between them is that Units are embedded with the references of the other Units to which they are connected (as indicated by the Unit 'Relation' characteristic). This is in contrast to Function Objects where connection data is stored separately in Connection Objects.

Therefore there would appear to be scope to apply the OCS data model to the basic Unit storage entity in order to separate a Unit's functionality from its connection information. This would result in the 'Binary Attribute', 'Attribute' and 'Behaviour' characteristics of each Unit being recorded within a single Function Object, and a Unit's 'Relation' characteristic being captured within one or more Connection Objects. The advantages gained would include more fine grain control over Storage Interface Units since Units would become re-usable and there would be greater opportunity for Unit versioning and improved maintenance.

12.7.7. Callimachus

Callimachus is a CB-OHS that provides an environment for the development and operation of structure servers [Tzagarakis et al. 2003]. Such structure servers are used to define structural models comprising new structural abstractions and services for new hypermedia domains. It is related to the work of the OCS data model as it too is concerned with the construction of hypermedia structure, and it utilises similar abstractions to OCS Function and Connection Objects.

One such similar abstraction is Structure Templates. They present patterns for structure that act as specifications of the structural model for a given hypermedia

domain. In this way Callimachus operates structure servers through being guided by Structure Templates which provide domain specific abstractions and constraints.

Structure Templates can be compared to OCS Connection Objects as both are used to assist with the dynamic building of structure. However the difference is with the role that each plays. Connection Objects describe how a *specific* structure is to be built, i.e. they identify which *specific* Function or Connection Objects are to be connected to one another. This is unlike Structure Templates which do not contain the connection information (between objects) to build actual structure. Structure Templates act more like classes in that Structure Templates guide the structure server as to which object *types* are permitted (and expected) to be attached to one another. For example specifying the number of Node objects that can connect to an Anchor object, or the number of destination Endpoints that can connect to a Link object. Hence unlike Connection Objects, Structure Templates are not capable of describing how a specific structure should be built.

Callimachus also employs Abstract Structural Elements. They are the basic structural primitive that a Structure Template uses to build hypermedia structure. Each Abstract Structural Element has a number of generic attributes that are programmed depending on the structural type that it is representing. One such attribute is endsets. An endset acts as a placeholder for recording the IDs of other structural objects. This enables related objects to be connected together in order to create high-level structural abstractions, such as navigational or taxonomic links. The Abstract Structural Element is an example implementation of Structural Computing's structural atom (Sections 2.15 and 12.7.4).

Abstract Structural Elements can be compared with OCS Function Objects since both form the basic block for building hypermedia structure. But a key difference between them is that the connection information between objects continues to be embedded within Callimachus' Abstract Structural Elements. As explained above, such information is recorded by the endset attribute within an Abstract Structural Element.

Therefore it can be concluded that Callimachus would benefit from application of the OCS data model to its Abstract Structural Elements. This would be in order to separate an Abstract Structural Element's function and connection information. However, the OCS data model might also benefit from Callimachus' implementation of Structure Templates. This is because a similar mechanism to Structure Templates can be used to declare the structural models that Connection Objects should adhere to when connecting Function and/or Connection Objects together.

12.7.8. Themis

Themis [Anderson et al. 2003a; Anderson et al. 2003b] provides a Structural Computing framework to facilitate the rapid construction of tools for a variety of hypermedia domains. Its focus is to reduce the size of the client application code that is responsible for creating instances of client application data structures. An example of an application data structure might be the layout of an address book containing structural labels such as name, address and phone number. Themis reduces application data structure code by removing such code from the applications themselves, and transforming it into structure templates which are recorded in structure servers. Thus the client can load the template from the Themis structure server, and use that template to create new instances of the data structure. Hence the client no longer needs to store the application structure data creation code.

An obvious comparison is between Themis Templates and OCS Connection Objects. This is because both are examples of structure being used to create structure. Themis Templates are used to create instances of structure, whilst Connection Objects are used to create the connections between objects within hypermedia structure. Moreover they adopt similar approaches to re-use. A Themis Template can reference (i.e. re-use) other templates as part of its definition, just the same as Connection Objects can re-use other Connection Objects when defining connections between hypermedia objects.

But the two structure mechanisms can also be regarded as fundamentally different to one another since they operate at different levels of structure. A Themis Template is like a class, i.e. it is waiting for user input in order to create an instance of the structure defined by the Template. This compares with a Connection Object, which is an example of a structure already instantiated with data. That data being the specific Function and/or Connection Objects that comprise the Connection Object content.

Themis also employs a conceptual element called the Themis Atom [Anderson et al. 2003a] which represents an atomic unit of structure. It can be compared with an OCS Function Object, because like OCS Function Objects, they are not embedded with connection data. Thus the same Atom can be grouped with different sets of Atoms. However Themis Atoms are not used in the same way as Function Objects. This is because Themis does not model the connections between Atoms, i.e. it does not have a mechanism like Connection Objects that model the connections between Function Objects. Instead Atoms are grouped together within collections. Hence, unlike the

OCS data model, Themis does not allow the connections between objects to be re-used.

12.8. Future Work

There are several areas that can be drawn out of the OCS data model that can be investigated as future work.

12.8.1. Implementation

The first activity to be undertaken would be an implementation of the OCS data model. A suitable starting point would be to test the OCS data model's approach upon actual OHP-Nav and FOHM structures. This is because, not only am I familiar with these two hypermedia models, but it was against these two data models that the operation and benefits of the OCS data model have been described within the thesis.

Other notable features to test include the impact that the OCS data model will have upon the number of hypermedia objects (Function and Connection Objects) that would be created and/or saved as a consequence of applying the OCS data model. Under assessment would also be the number of additional messages necessary to be sent around the network in order to store, build and retrieve hypermedia structure. This would be coupled with analysis of any added complexity that the OCS data model would bring when creating overall top-levels of structure, such as hypertext links.

Also to be investigated would be the effect the OCS data model would have upon hypermedia object re-usability, hypermedia object versioning and hypermedia object referential integrity. This would reveal to what extent the general manipulation and management of individual hypermedia objects would be improved through application of the OCS data model.

12.8.2. Data Models

Thus far the impact of the OCS data model has been considered in relation to two data models, OHP-Nav and FOHM. Therefore a natural avenue for future research is to explore application of the OCS model to other hypermedia structure data models, e.g. the HURL hypermedia model [Hicks et al. 1998].

12.8.3. Hypermedia Domains

The OCS data model throughout the thesis has primarily concentrated on the navigational domain. But the OCS data model can just as equally be applied to other hypermedia domains as well, e.g. the taxonomic domain.

This is because the taxonomic domain records relationships for hierarchical classification schemes, but often it cannot be decided to which classification an object should belong. Hence objects are often assigned to more than one hierarchical structure. This is where the OCS data model can be useful as it enables objects to be re-used between structures. Firstly this would enable more economical object storage. Secondly, by re-using the same object between two or more structures, it gives an immediate indication of a shared relationship between those structures. And finally, it highlights the undetermined relationship about the classification finality of the object between the structures.

12.8.4. The OCS Framework

The OCS framework concept (described in Chapter 10) can be expanded beyond versioning just structure. For example, it can be used as the basis for connecting documents within hypermedia networks. In which case documents would be represented by Function Objects and hypertext links by Connection Objects.

12.8.5. Standardisation of Connection Objects

The OCS data model can significantly benefit the OHSWG standardisation effort. Rather than focusing just on the standardisation on generalised hypermedia objects, the OHSWG should also concentrate their efforts on standardising OCS Connection Objects. OHP is comprised of many hypermedia object types where each object type has a different functional role. It has been because of the range of functions that an object may perform that has resulted in the OHSWG taking a long time to decide which objects should form the structure of the OHP. The problem is that there is a potentially unlimited range of functionality that the OHP can address. However, what all hypermedia objects have in common is that they are all connected to other objects, i.e. they all have some form of connection data. Therefore, it is also this connectional aspect of structure that should be standardised separate to an object's functional aspect.

12.8.6. Link Maintenance Repair Applications

The maintenance of generic structure connections can also benefit from the standardisation of OCS Connection Objects. This is because if all connections between objects are standardised, then this will ease the process of creating structure repair applications. The knock-on effect is that it should encourage development of structure repair applications. Thus if more structure repair applications are available, then users will be more likely to adopt them for maintaining their structure.

12.9. Overall Conclusion

Connection data should not be mixed in with the function data of a hypermedia object. This is the basic idea behind the OCS data model. The result is that separating a hypermedia object's function and connection data can spawn many benefits. Not least the opening up of hypermedia structure to enable re-usability, improved structure versioning and structure maintenance. This is evidenced by the OCS data model meeting the 4 Problem Domain Issues and achieving the 5 Computer Science Contributions.

However there is some cost in adopting the OCS data model. Whilst the OCS data model significantly benefits the organisation of structure it also adds to its complexity (Section 7.16). And it is often the case that more objects are generated when adopting the OCS data model. But these are often only lightweight Connection Objects. Of more significance is that the OCS data model typically reduces the number of heavyweight Function Objects (Section 7.9).

Taken as a whole, the OCS data model has a positive impact for computer users and the general computing environment. It eases structure creation, enables discovery of new relationships, reduces message sizes, assists hypermedia domain interoperability and offers optimization at the storage level (Section 7.15). It also assists the OHSWG by directing them to concentrate their standardisation efforts on the connections between generic hypermedia objects.

Appendix A. OCS Representation in XML

A.1. Introduction

This appendix describes one possible XML representation for OCS Function and Connection Objects. It centres on a simplified form of the original Darmstadt OHP-Nav XML specification [Reich and Millard 1999] which focuses on the four primary objects: Node, Anchor, Endpoint and Link. It does not include PSpec or Context objects as these are unnecessary for basic hypertext link construction and presentation.

A.2. Conventional OHP-Nav Primary Objects

This section lists the simplified form of the conventional XML definitions of the Node, Anchor, Endpoint and Link objects used by the Solent CB-OHS.

```
<!-- Simplified NODE element.
     Does not include Computation or PSpec elements. -->
<!ELEMENT NODE (ID, NAME?, CONTENTSPEC, (%abObjInfo;))>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT CONTENTSPEC (URL?, CONTENT?, CHARACTERISTICSET?,
          VERSION?, MIMETYPE?)>
<!ELEMENT URL (#PCDATA)>
<!ELEMENT VERSION (#PCDATA)>
<!ELEMENT CONTENT (#PCDATA)>
<!ELEMENT MIMETYPE (#PCDATA)>
<!ENTITY  %abObjInfo "MYTYPE?, DESCRIPTIONSET?,
          CHARACTERISTICSET?">
```

```
<!ELEMENT MYTYPE  (#PCDATA)>

<!ELEMENT DESCRIPTIONSET (DESCRIPTION*)>

<!ELEMENT DESCRIPTION (NAME, VALUE)>

<!ELEMENT NAME (#PCDATA)>

<!ELEMENT VALUE (#PCDATA)>

<!ELEMENT CHARACTERISTICSET (#PCDATA)>


<!-- Simplified ANCHOR element.
     Includes the locSpec entity as just the AXISLOC element.
     Does not include Computation or PSpec elements -->

<!ELEMENT ANCHOR  (ID, NAME?, PARENTID, AXISLOC,
          (%abObjInfo;))>

<!ELEMENT PARENTID (#PCDATA)>

<!ELEMENT AXISLOC (FWDAXISSET, REVAXISSET?, VERSION, ID)>

<!ELEMENT FWDAXISSET (AXIS*)>

<!ELEMENT REVAXISSET (AXIS*)>

<!ELEMENT AXIS (NAME?, TYPE, VALUESET)>

<!ELEMENT TYPE (#PCDATA)>

<!ELEMENT VALUESET (VALUE*)>


<!-- Simplified ENDPOINT element.
     Does not include Computation or PSpec elements. -->

<!ELEMENT ENDPOINT (ID, NAME?, DIRECTION, (%abObjInfo),
          ANCHORID)>

<!ELEMENT DIRECTION (#PCDATA)>

<!ELEMENT ANCHORID (#PCDATA)>


<!-- Simplified LINK element.
     Does not include Computation or PSpec elements. -->

<!ELEMENT LINK (ID, NAME?, (%abObjInfo), ENDPOINTIDSET?)>

<!ELEMENT ENDPOINTIDSET (ID, ID*)>
```

A.3. OCS Function Objects

OCS Function Objects are essentially the same as conventional OHP-Nav objects (Section A.2). The only difference being that they do not contain embedded references to other Function Objects.

As was the case with the conventional representation of OHP-Nav (Section A.2), the example XML representation of OCS Function Objects also does not include PSpec or Context objects. The XML simplified definitions of Node, Anchor, Endpoint and Link objects are listed.

```
<!-- Simplified OCS Function NODE element.

     Does not include Computation or PSpec elements.

     This element is the same as a conventional Node element
     since it does not contain any embedded object references.

     All elements within an OCS NODE are the same as defined
     in Section A.2. -->

<!ELEMENT NODE (ID, NAME?, CONTENTSPEC, (%abObjInfo;))>


<!-- Simplified OCS Function ANCHOR element.

     Includes the locSpec entity as just the AXISLOC element.

     Does not include Computation or PSpec elements.

     Note that the PARENTID element is omitted since it is a
     form of object reference embedding.

     All elements within an OCS ANCHOR are the same as defined
     in Section A.2. -->

<!ELEMENT ANCHOR  (ID, NAME?, AXISLOC, (%abObjInfo;))>


<!-- Simplified OCS Function ENDPOINT element.

     Does not include Computation or PSpec elements.

     Note that the ANCHORID element is omitted since it is a
     form of object reference embedding.

     All elements within an OCS Endpoint are the same as
     defined in Section A.2. -->

<!ELEMENT ENDPOINT (ID, NAME?, DIRECTION, (%abObjInfo))>
```

```
<!-- Simplified LINK element.

     Does not include Computation or PSpec elements.

     Note that the ENDPOINTIDSET element is omitted since it
     is a form of object reference embedding.

     All elements within an OCS Link object are the same as
     defined in Section A.2. -->

<!ELEMENT LINK (ID, NAME?, (%abObjInfo))>
```

A.4. OCS Connection Objects

OCS Connection Objects record the connections between OCS Function Objects.

This XML representation enables basic Connection Object manipulation to occur. It should be remembered that it is only an example XML representation, other XML representations of Connection Objects are possible and perfectly acceptable.

```
<!-- Connection Object -->

<!ELEMENT CONNECTION_OBJECT (ID, CONNECTION_LIST?)>


<!ELEMENT ID (#PCDATA)>

<!ELEMENT CONNECTION_LIST (CONNECTION*)>

<!ELEMENT CONNECTION (OBJECT*)>


<!-- An OBJECT can be a Function Object or a Connection
     Object. If bond entity is missing, then it is presumed to
     be an ATTACH bonding operation. -->

<!ELEMENT OBJECT (BOND?, OBJECT_TYPE?, ID, INSTANCE_ID?,
         INT_ANCHOR?)>


<!-- BOND can join together Connection Object to Connection
     Object, Connection Object to Function Object (and vice
     versa), or Function Object to Function Object. Both
     Connection Objects and Function Objects are identified
     via OBJECT element.

     BOND can produce new Connection Objects or replace
     existing Connection Objects. To produce a new Connection
     Object, then use a new ID or do not assign any ID at the
     CONNECTION_OBJECT level. To replace an existing
     Connection Object, i.e. so one of the Connection Objects
```

```
     being bonded takes on the role of the entire bonded
     Connection Object, then use their existing ID at the
     CONNECTION_OBJECT level. -->

<!ENTITY % BOND "ATTACH | CONJOIN">


<!-- OBJ_TYPE used for clarity in defining what type of object
     (either a Function or Connection Object) is being
     described. -->

<!ENTITY % OBJ_TYPE "FUNCTION | CONNECTION">


<!-- An OBJECT is identified as an Instance Object if it has
     an INSTANCE_ID. -->

<!ELEMENT INSTANCE_ID (#PCDATA)>


<!-- INT_ANCHOR is an internal anchor within a Connection
     Object, although it can be used for Function Objects, but
     these are assumed to be opaque. -->

<!ELEMENT INT_ANCHOR (OBJECT)>


<!-- Operations that can be enacted on Connection Objects. -->


<!-- Creating a Connection Object. -->

<!ELEMENT CONNECTION_OBJECT_CREATE (CONNECTION_OBJECT)>

<!ELEMENT CONNECTION_OBJECT_CREATED (CONNECTION_OBJECT)>


<!-- Deleting a Connection Object. -->

<!ELEMENT CONNECTION_OBJECT_DELETE (ID)>

<!ELEMENT CONNECTION_OBJECT_DELETED (ID)>


<!-- Modifying a Connection Object. -->

<!ELEMENT CONNECTION_OBJECT_MODIFY (ID,
          CONNECTION_OBJECT_TEMPLATE, MODTYPE)>

<!ELEMENT CONNECTION_OBJECT_MODIFIED (ID)>

<!ELEMENT CONNECTION_OBJECT_TEMPLATE (ID?, CONNECTION_LIST?)>

<!ENTITY % modType "REMOVE | CREATE | ADD | DELETE">


<!-- Retrieving a Connection Object. -->
```

```
<!ELEMENT CONNECTION_OBJECT_RETRIEVE (ID,
          CONNECTION_OBJECT_TEMPLATE?)>

<!ELEMENT CONNECTION_OBJECT_RETRIEVED (CONNECTION_OBJECT)>
```

A.5. Re-use Example

This section lists the XML code to describe an Object and Connection Space re-use example. Figure A.1 displays the Function Object Space, and Figures A.2 and A.3 display the Connection Spaces.

This example shows how the OCS data model enables the same Function Objects to be re-used to produce different hypermedia structures in order to portray different relationships. The example also shows the use of Instance Objects, in this case Anchor object A1, being re-used within Connection Object C500.

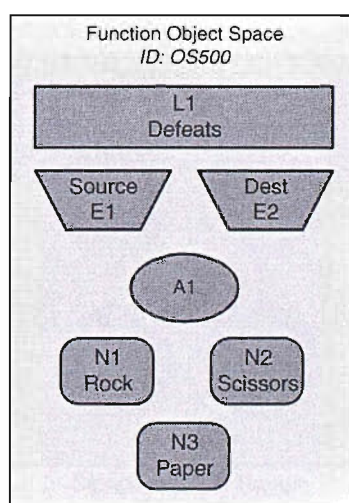


Figure A.1: Function Object Space listing all Function Objects required for all RPS 'Defeats' scenarios.

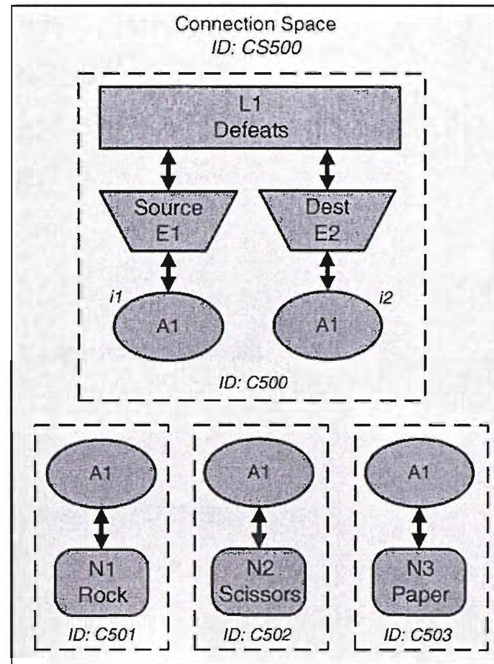


Figure A.2: Connection Space CS500.

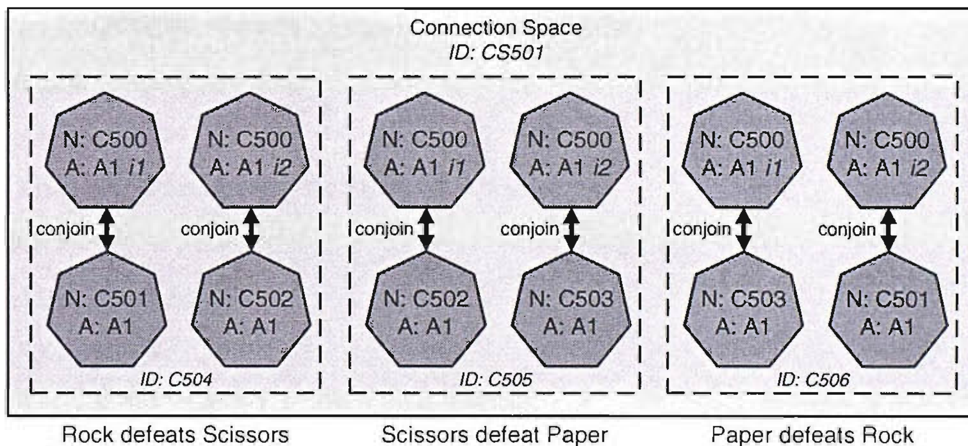


Figure A.3: Connection Space facilitating all RPS 'Defeats' scenarios.

A.5.1. Function Object Space

Here are the separate Function Objects of Figure A.1.

```
<!-- Link Function Object L1 -->
<LINK>
   <ID> L1 </ID>
   <DESCRIPTIONSET>
       <DESCRIPTION>
```

```
        <NAME> type </NAME>
        <VALUE> defeats </VALUE>
      </DESCRIPTION>
   </DESCRIPTIONSET>
</LINK>

<!-- Endpoint Function Object E1 -->
<ENDPOINT>
    <ID> E1 </ID>
    <DIRECTION> source </DIRECTION>
</ENDPOINT>

<!-- Endpoint Function Object E2 -->
<ENDPOINT>
    <ID> E2 </ID>
    <DIRECTION> destination </DIRECTION>
</ENDPOINT>

<!-- Anchor Function Object A1 -->
<ANCHOR>
    <ID> A1 </ID>
    <AXISLOC>
        <FWDAXISSET>
           <AXIS>
             <TYPE> whole </TYPE>
             <VALUESET/>
           </AXIS>
        </FWDAXISSET>
    </AXISLOC>
</ANCHOR>

<!-- Node Function Object N1 -->
<NODE>
    <ID> N1 </ID>
        <CONTENTSPEC>
```

```
        <URL> C:\rock.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>

<!-- Node Function Object N2 -->
<NODE>
    <ID> N2 </ID>
    <CONTENTSPEC>
        <URL> C:\scissors.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>

<!-- Node Function Object N3 -->
<NODE>
    <ID> N3 </ID>
    <CONTENTSPEC>
        <URL> C:\paper.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>
```

A.5.2. Connection Object Space CS500

Here are the Connection Objects of the Connection Space of Figure A.2.

```
<!-- C500: A single Connection Object expressing the 'Defeats'
     relationship. -->
<CONNECTION_OBJECT>
    <ID> C500 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
           <OBJECT>
              <ID> L1 </ID>
```

```
          </OBJECT>
          <OBJECT>
            <ID> E1 </ID>
          </OBJECT>
        </CONNECTION>
        <CONNECTION>
          <OBJECT>
            <ID> L1 </ID>
          </OBJECT>
          <OBJECT>
            <ID> E2 </ID>
          </OBJECT>
        </CONNECTION>
        <CONNECTION>
          <OBJECT>
            <ID> E1 </ID>
          </OBJECT>
          <OBJECT>
            <ID> A1 </ID>
            <INSTANCE_ID> 01 </INSTANCE_ID>
          </OBJECT>
        </CONNECTION>
        <CONNECTION>
          <OBJECT>
            <ID> A1 </ID>
            <INSTANCE_ID> 02 </INSTANCE_ID>
          </OBJECT>
          <OBJECT>
            <ID> E2 </ID>
          </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>
```

```
<!-- C501: Connection Object to connect A1 to N1 (Rock). -->
<CONNECTION_OBJECT>
    <ID> C501 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
            <OBJECT>
              <ID> N1 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>

<!-- C502: Connection Object to connect A1 to N2 (Scissors).
     -->
<CONNECTION_OBJECT>
    <ID> C502 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
            <OBJECT>
              <ID> N2 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>

<!-- C503: Connection Object to connect A1 to N3 (Paper). -->
<CONNECTION_OBJECT>
    <ID>C503</ID>
    <CONNECTION_LIST>
```

```
        <CONNECTION>
            <OBJECT>
                <ID> A1 </ID>
            </OBJECT>
            <OBJECT>
                <ID> N3 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>
```

A.5.3. Connection Object Space CS501

The three Connection Objects of the Connection Space of Figure A.3 together create the three hyperstructure relationships: 'Rock defeats Scissors', 'Scissors defeat Paper' and 'Paper defeats Rock'.

```
<!-- C504: Connection Object that creates 'Rock defeats
     Scissors'.
     It connects A1 (instance 01) of C500 to A1 of C501
     (Rock); and connects A1 (instance 02) of C500 to A1 of
     C502 (Scissors). -->
<CONNECTION_OBJECT>
    <ID> C504 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <BOND> CONJOIN </BOND>
            <OBJECT>
                <ID> C500 </ID>
                <INT_ANCHOR>
                    <OBJECT>
                        <ID> A1 </ID>
                        <INSTANCE_ID> 01 </INSTANCE_ID>
                    </OBJECT>
                </INT_ANCHOR>
```

```
        </OBJECT>
        <OBJECT>
          <ID> C501 </ID>
          <INT_ANCHOR>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
          </INT_ANCHOR>
        </OBJECT>
      </CONNECTION>
      <CONNECTION>
        <BOND> CONJOIN </BOND>
        <OBJECT>
          <ID> C500 </ID>
          <INT_ANCHOR>
            <OBJECT>
              <ID> A1 </ID>
              <INSTANCE_ID> 02 </INSTANCE_ID>
            </OBJECT>
          </INT_ANCHOR>
        </OBJECT>
        <OBJECT>
          <ID> C502 </ID>
          <INT_ANCHOR>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
          </INT_ANCHOR>
        </OBJECT>
      </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>
```

```
<!-- C505: Connection Object that creates 'Scissors defeat
     Paper'.
     It Connects A1 (instance 01) of C500 to A1 of C502
     (Scissors); and connects A1 (instance 02) of C500 to A1
     of C503 (Paper). -->
<CONNECTION_OBJECT>
    <ID> C505 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <BOND> CONJOIN </BOND>
            <OBJECT>
              <ID> C500 </ID>
              <INT_ANCHOR>
                <OBJECT>
                  <ID> A1 </ID>
                  <INSTANCE_ID> 01 </INSTANCE_ID>
                </OBJECT>
              </INT_ANCHOR>
            </OBJECT>
            <OBJECT>
              <ID> C502 </ID>
              <INT_ANCHOR>
                <OBJECT>
                  <ID> A1 </ID>
                </OBJECT>
              </INT_ANCHOR>
            </OBJECT>
        </CONNECTION>
        <CONNECTION>
            <BOND> CONJOIN </BOND>
            <OBJECT>
              <ID> C500 </ID>
              <INT_ANCHOR>
                <OBJECT>
                  <ID> A1 </ID>
                  <INSTANCE_ID> 02 </INSTANCE_ID>
```

```
            </OBJECT>
          </INT_ANCHOR>
        </OBJECT>
        <OBJECT>
          <ID> C503 </ID>
          <INT_ANCHOR>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
          </INT_ANCHOR>
        </OBJECT>
      </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>

<!-- C506: Connection Object that creates 'Paper defeat Rock'.
     It connects A1 (instance 01) of C500 to A1 of C503
     (Paper); and connects A1 (instance 02) of C500 to A1 of
     C501 (Rock). -->
<CONNECTION_OBJECT>
    <ID> C506 </ID>
    <CONNECTION_LIST>
      <CONNECTION>
        <BOND> CONJOIN </BOND>
        <OBJECT>
          <ID> C500 </ID>
          <INT_ANCHOR>
            <OBJECT>
              <ID> A1 </ID>
              <INSTANCE_ID> 01 </INSTANCE_ID>
            </OBJECT>
          </INT_ANCHOR>
        </OBJECT>
        <OBJECT>
          <ID> C503 </ID>
```

```
            <INT_ANCHOR>
              <OBJECT>
                <ID> A1 </ID>
              </OBJECT>
            </INT_ANCHOR>
          </OBJECT>
        </CONNECTION>
        <CONNECTION>
          <BOND> CONJOIN </BOND>
          <OBJECT>
            <ID> C500 </ID>
            <INT_ANCHOR>
              <OBJECT>
                <ID> A1 </ID>
                <INSTANCE_ID> 02 </INSTANCE_ID>
              </OBJECT>
            </INT_ANCHOR>
          </OBJECT>
          <OBJECT>
            <ID> C501 </ID>
            <INT_ANCHOR>
              <OBJECT>
                <ID> A1 </ID>
              </OBJECT>
            </INT_ANCHOR>
          </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>
```

A.6. Converting to OCS Data Model Format Example

This section lists the XML code for the scenario depicted by Figure 8.4. It provides an example as to how the OCS data model can be used to imitate the conventional representation of OHP-Nav hypermedia structure. For ease, Figure 8.4 has been reproduced here as Figures A.4, A.5 and A.6.

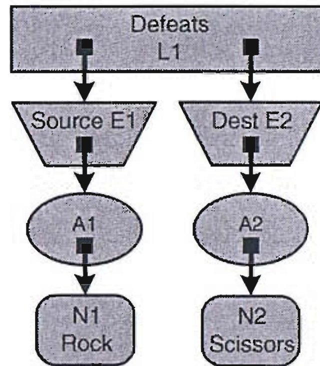


Figure A.4: Conventional OHP-Nav embedded object referencing hypertext link representation of Figure 8.4.

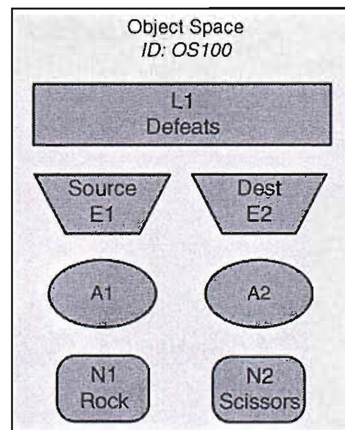


Figure A.5: OCS Function Objects imitating conventional OHP-Nav objects.

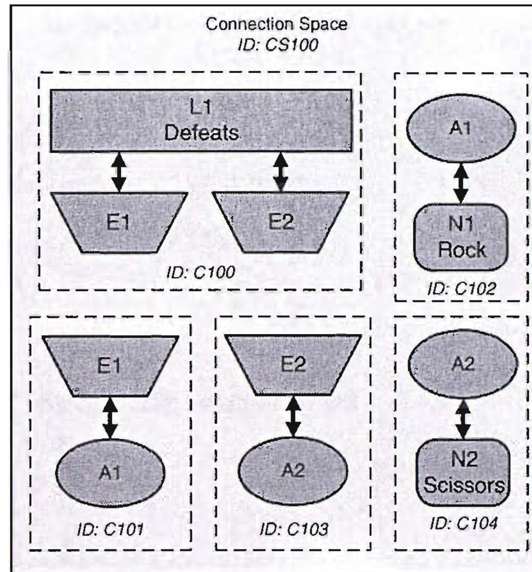


Figure A.6: OCS Connection Objects imitating conventional OHP-Nav connections.

A.6.1. XML for Conventional Hypertext Link

This subsection lists the XML code necessary for the conventional embedded object referencing hypertext link of Figure A.4.

```
<!-- Conventional Link object L1. -->
<LINK>
    <ID> L1 </ID>
    <DESCRIPTIONSET>
        <DESCRIPTION>
           <NAME> type </NAME>
           <VALUE> defeats </VALUE>
        </DESCRIPTION>
    </DESCRIPTIONSET>
    <ENDPOINTIDSET>
        <ID> E1 </ID>
        <ID> E2 </ID>
    </ENDPOINTIDSET>
</LINK>
```

```
<!-- Conventional Endpoint object E1. -->
<ENDPOINT>
    <ID> E1 </ID>
    <DIRECTION> source </DIRECTION>
    <ANCHORID> A1 </ANCHORID>
</ENDPOINT>

<!-- Conventional Endpoint object E2. -->
<ENDPOINT>
    <ID> E2 </ID>
    <DIRECTION> destination </DIRECTION>
    <ANCHORID> A2 </ANCHORID>
</ENDPOINT>

<!-- Conventional Anchor object A1. -->
<ANCHOR>
    <ID> A1 </ID>
    <PARENTID> N1 </PARENTID>
    <AXISLOC>
        <FWDAXISSET>
           <AXIS>
             <NAME> characters in </NAME>
             <TYPE> CHAR </TYPE>
             <VALUESET>
               <VALUE> 29 </VALUE>
               <VALUE> 38 </VALUE>
             </VALUESET>
           </AXIS>
        </FWDAXISSET>
    </AXISLOC>
</ANCHOR>

<!-- Conventional Anchor object A2. -->
<ANCHOR>
    <ID> A2 </ID>
```

```
    <PARENTID> N2 </PARENTID>
    <AXISLOC>
        <FWDAXISSET>
            <AXIS>
                <NAME> characters in </NAME>
                <TYPE> CHAR </TYPE>
                <VALUESET>
                    <VALUE> 50 </VALUE>
                    <VALUE> 57 </VALUE>
                </VALUESET>
            </AXIS>
        </FWDAXISSET>
    </AXISLOC>
</ANCHOR>

<!-- Conventional Node object N1. -->
<NODE>
    <ID> N1 </ID>
    <CONTENTSPEC>
        <URL> C:\rock.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>

<!-- Conventional Node object N2. -->
<NODE>
    <ID> N2 </ID>
    <CONTENTSPEC>
        <URL> C:\scissors.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>
```

A.6.2. XML for OCS Function Objects

This subsection lists the XML code for the Function Objects (shown by figure A.5) when the OHP-Nav hypertext link is converted to the OCS data model.

```
<!-- OCS Link Function Object L1. -->
<LINK>
    <ID> L1 </ID>
    <DESCRIPTIONSET>
        <DESCRIPTION>
           <NAME> type </NAME>
           <VALUE> defeats </VALUE>
        </DESCRIPTION>
    </DESCRIPTIONSET>
</LINK>

<!-- OCS Endpoint Function Object E1. -->
<ENDPOINT>
    <ID> E1 </ID>
    <DIRECTION> source </DIRECTION>
</ENDPOINT>

<!-- OCS Endpoint Function Object E2. -->
<ENDPOINT>
    <ID> E2 </ID>
    <DIRECTION> destination </DIRECTION>
</ENDPOINT>

<!-- OCS Anchor Function Object A1. -->
<ANCHOR>
    <ID> A1 </ID>
    <AXISLOC>
        <FWDAXISSET>
           <AXIS>
             <NAME> characters in </NAME>
```

```
            <TYPE> CHAR </TYPE>
            <VALUESET>
              <VALUE> 29 </VALUE>
              <VALUE> 38 </VALUE>
            </VALUESET>
          </AXIS>
        </FWDAXISSET>
    </AXISLOC>
</ANCHOR>

<!-- OCS Anchor Function Object A2. -->
<ANCHOR>
    <ID> A2 </ID>
    <AXISLOC>
        <FWDAXISSET>
          <AXIS>
            <NAME> characters in </NAME>
            <TYPE> CHAR </TYPE>
            <VALUESET>
              <VALUE> 50 </VALUE>
              <VALUE> 57 </VALUE>
            </VALUESET>
          </AXIS>
        </FWDAXISSET>
    </AXISLOC>
</ANCHOR>

<!-- OCS Node Function Object N1. -->
<NODE>
    <ID> N1 </ID>
    <CONTENTSPEC>
        <URL> C:\rock.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>
```

```
<!-- OCS Node Function Object N2. -->
<NODE>
    <ID> N2 </ID>
    <CONTENTSPEC>
        <URL> C:\scissors.html </URL>
        <MIMETYPE> text/html </MIMETYPE>
    </CONTENTSPEC>
</NODE>
```

A.6.3. XML for OCS Connection Objects

This subsection lists the XML code for the Connection Objects shown by Figure A.6 that imitate the connections within the conventional hypertext link of Figure A.4.

```
<!-- OCS Link Function Object L1 connected to OCS Endpoint
     Function Objects E1 and E2. -->
<CONNECTION_OBJECT>
    <ID> C100 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
                <ID> L1 </ID>
            </OBJECT>
            <OBJECT>
                <ID> E1 </ID>
            </OBJECT>
        </CONNECTION>
        <CONNECTION>
            <OBJECT>
                <ID> L1 </ID>
            </OBJECT>
            <OBJECT>
                <ID> E2 </ID>
```

```
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>

<!-- OCS Endpoint Function Object E1 connected to OCS Anchor
     Function Object A1. -->
<CONNECTION_OBJECT>
    <ID> C101 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
              <ID> E1 </ID>
            </OBJECT>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>

<!-- OCS Endpoint Function Object E2 connected to OCS Anchor
     Function Object A2. -->
<CONNECTION_OBJECT>
    <ID> C103 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
              <ID> E2 </ID>
            </OBJECT>
            <OBJECT>
              <ID> A2 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
```

```
</CONNECTION_OBJECT>

<!-- OCS Anchor Function Object A1 connected to OCS Node
     Function Object N1. -->
<CONNECTION_OBJECT>
    <ID> C102 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
              <ID> A1 </ID>
            </OBJECT>
            <OBJECT>
              <ID> N1 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>

<!-- OCS Anchor Function Object A2 connected to OCS Node
     Function Object N2. -->
<CONNECTION_OBJECT>
    <ID> C104 </ID>
    <CONNECTION_LIST>
        <CONNECTION>
            <OBJECT>
              <ID> A2 </ID>
            </OBJECT>
            <OBJECT>
              <ID> N2 </ID>
            </OBJECT>
        </CONNECTION>
    </CONNECTION_LIST>
</CONNECTION_OBJECT>
```

Appendix B. Examples of OCS Re-use

B.1. Introduction

This appendix section follows on from Section 8.4. It describes how the OCS data model resolves the remaining two scenarios outlined in Sections 6.2 and 6.3. These are enabling the re-use of single FOHM objects (Section B.2) and the re-use of whole segments of OHP-Nav structure (Section B.3).

B.2. FOHM Single Hypermedia Object Re-use

Section 6.2.2 explains how the FOHM data model prevents the re-use of the functionally identical objects within the two FOHM structures of Figure 6.3.

As reported in Section 8.3 the OCS data model leaves the decision on whether to embed whole objects within one another as a design choice for OCS hypermedia structure creators. Therefore Section B.2.1 provides an OCS solution that includes wholesale object embedding and Section B.2.2 provides an OCS solution that precludes wholesale object embedding.

B.2.1. OCS Solution with FOHM Object Embedding

Figure B.1 shows the OCS data model solution that allows wholesale object embedding. This is coupled with Table B.1 which summarises the re-use taking place within this OCS solution.

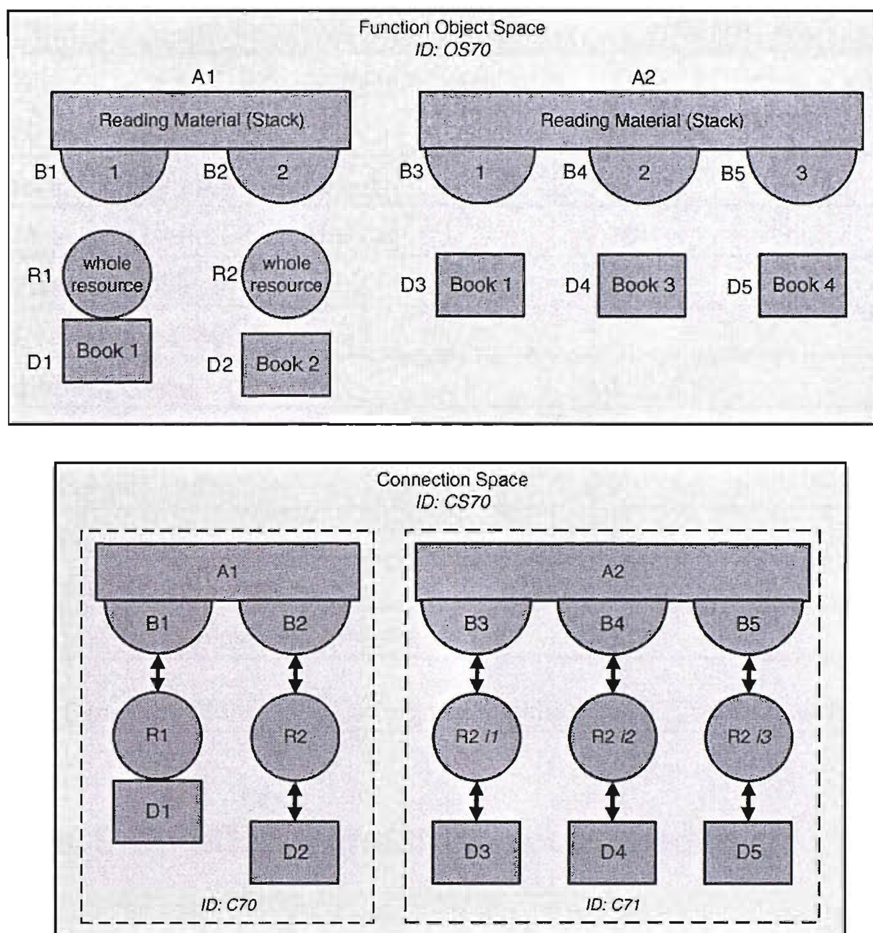


Figure B.1: OCS solution enabling single FOHM object re-use for the FOHM structure of Figure 6.3 that permits wholesale object embedding.

The OCS solution produces an overall total of 10 objects. This is a saving of 1 object over the original structure of Figure 6.3. However 2 of the OCS solution objects are Connection Objects. Because these are considered lightweight objects, then there is actually a greater saving of 3 equivalent hypermedia objects over the original structure of Figure 6.3. Table B.1 records the eliminated objects as R3, R4 and R5. They have been replaced by a single copy of R2 within Connection Object C71.

Description	Value
No. of objects to build original FOHM structures (using object embedding)	11
No. of functionally identical objects	4
No. of objects to build OCS solution	10
No. of Function Objects	8
No. of Connection Objects	2
Re-used object count	1
Re-used object list	R2 for R3, R4, R5
Eliminated object count	3
Eliminated object list	R3, R4, R5
No. of overall objects saved	1
No. of equivalent hypermedia objects saved	3

Table B.1: Summary of the re-use taking place within the OCS solution of Figure B.1.

B.2.2. OCS Solution without Object Embedding

This second OCS solution does not allow wholesale object embedding. In this scenario the embedded objects are separated from their container objects. I.e. B1 and B2 are separated from A1; B3, B4 and B5 are separated from A2; and D1 is separated from R1. Hence the original structure of Figure 6.3 can be perceived as containing a total of 17 objects. Figure B.2 shows the OCS solution together with Table B.2 which presents a summary of the re-use occurring within the OCS solution.

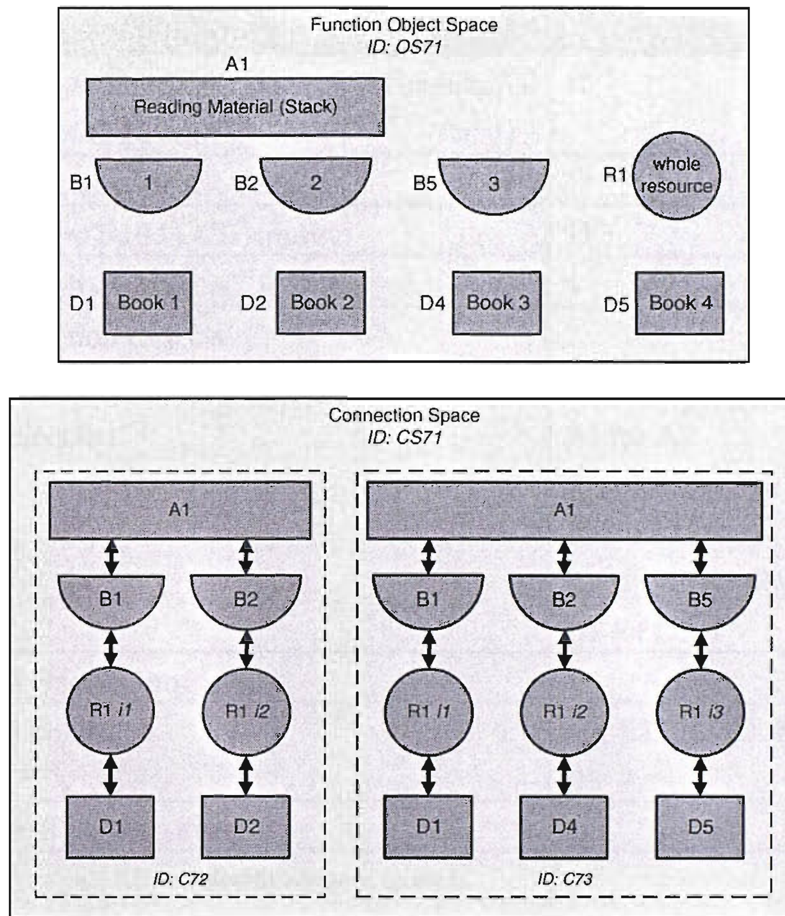


Figure B.2: OCS solution enabling single FOHM object re-use for the FOHM structure of Figure 6.3 without utilising object embedding.

This OCS solution demonstrates that separating embedded objects from their containing objects allow greater single object re-use. For example:

- A1 no longer contains embedded Bindings B1 and B2. Hence A1 can now be re-used in place of A2.

- Bindings B1, B2, B3, B4 and B5 have been dis-embedded from Associations A1 and A2. This enables B1 to be re-used in place of B3, and B2 to be re-used in place of B4.

- Data object D1 has been separated from Reference R1. This allows R1 (now without D1) to be re-used in place of References R2, R3, R4 and R5. And it also allows D1 (now a separate object) to be re-used in place of D3.

Description	Value
No. of objects to build original FOHM structures (without object embedding)	17
No. of functionally identical objects	13
No. of objects to build OCS solution	11
No. of Function Objects	9
No. of Connection Objects	2
Re-used object count	8
Re-used object list	A1 for A2 B1 for B3 B2 for B4 R1 for R2, R3, R4, R5 D1 for D3
Eliminated object count	8
Eliminated object list	A2, B3, B4, R2, R3, R4, R5, D3
No. of overall objects saved	6
No. of equivalent hypermedia objects saved	8

Table B.2: Summary of the re-use taking place within the OCS solution of Figure B.2.

Table B.2 shows that this OCS solution uses a total of 11 objects. This compares with the 17 objects that are needed to construct the two FOHM structures if built without FOHM object embedding. Hence the OCS solution saves 6 objects overall, and 8 equivalent hypermedia objects if excluding Connection Objects (because they are considered lightweight objects).

The total 11 OCS solution objects also equals the total number of objects used to construct the two FOHM structures if built using FOHM object embedding (as described in Section 6.2.2). But if the 2 Connection Objects are excluded, then there is actually a saving of 2 equivalent hypermedia objects over the original object-embedded structure of Figure 6.3. Moreover this approach of separating embedded objects from their containers also offers more opportunities for re-use in the future since more objects are available from which to select for re-use.

B.3. OHP-Nav Repetitive Hypermedia Structure Re-use

Section 6.3.1 explains how OHP-Nav object reference embedding prevents the general re-use of hypermedia structure segments. The two scenarios described are the 'Cats Eat' something relationship and the Node-less 'Eat' relationship.

B.3.1. 'Cats Eat' Something Relationship

Section 6.3.1 explains how the conventional OHP-Nav object reference embedding approach prevents the highlighted structure segment of Hyperlink A from being re-used in place of the highlighted segment of Hyperlink B within Figure 6.4.

Adopting the OCS data model allows such structure segment re-use to take place. The OCS solution is shown in Figure B.3 along with Table B.3 which summarises the object re-use within the OCS solution.

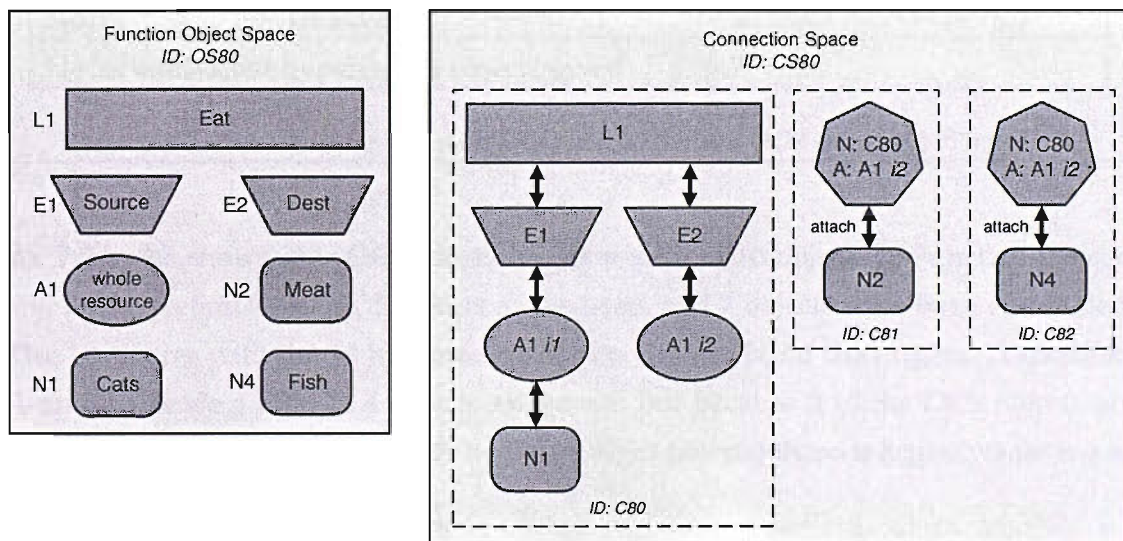


Figure B.3: OCS solution enabling structure segment re-use for the OHP-Nav 'Cats Eat' something relationship of Figure 6.4.

This OCS solution uses Function Objects N1, A1, E1, L1 and E2 to create the highlighted structure segment of Hyperlink A (the 'Cats Eat' something relationship). Together these Function Objects are recorded as single Connection Object C80. This enables that segment to be re-used within any hypermedia structure. This is demonstrated by Connection Objects C81 and C82 which re-use the highlighted segment C80 to build Hyperlinks A and B respectively.

Description	Value
No. of objects to build original OHP-Nav structures	14
No. of functionally identical objects	12
No. of objects to build OCS solution	10
No. of Function Objects	7
No. of Connection Objects	3
Re-used object count	5
Re-used object list	N1 for N3 A1 for A2, A3, A4 E1 for E3 L1 for L2 E2 for E4
Eliminated object count	7
Eliminated object list	N3, A2, A3, A4, E3, L2, E4
No. of overall objects saved	4
No. of equivalent hypermedia objects saved	7

Table B.3: Summary of the re-use taking place within the OCS solution of Figure B.3.

As Table B.3 shows, this OCS solution uses a total of 10 objects (7 Function Objects and 3 Connection Objects), 5 objects are re-used, and 7 objects have been eliminated. This compares with the 14 hypermedia objects used to build the original Hyperlinks A and B. Hence a total of 4 objects are saved. But because 3 of the OCS objects are Connection Objects (considered to be lightweight objects) there is actually a saving of 7 equivalent hypermedia objects.

A further benefit with adopting the OCS data model is that it enables Hyperlinks A and B to remain as two separate structures even though they share the same structure segment. This is possible because each structure is represented by different Connection Objects, i.e. C81 and C82. Hence when the two structures are constructed they are built as separate structures. Figure B.4 shows the hypermedia structures as a result of instantiating C81 and C82, i.e. Hyperlinks A and B sharing the same hypermedia structure segment.

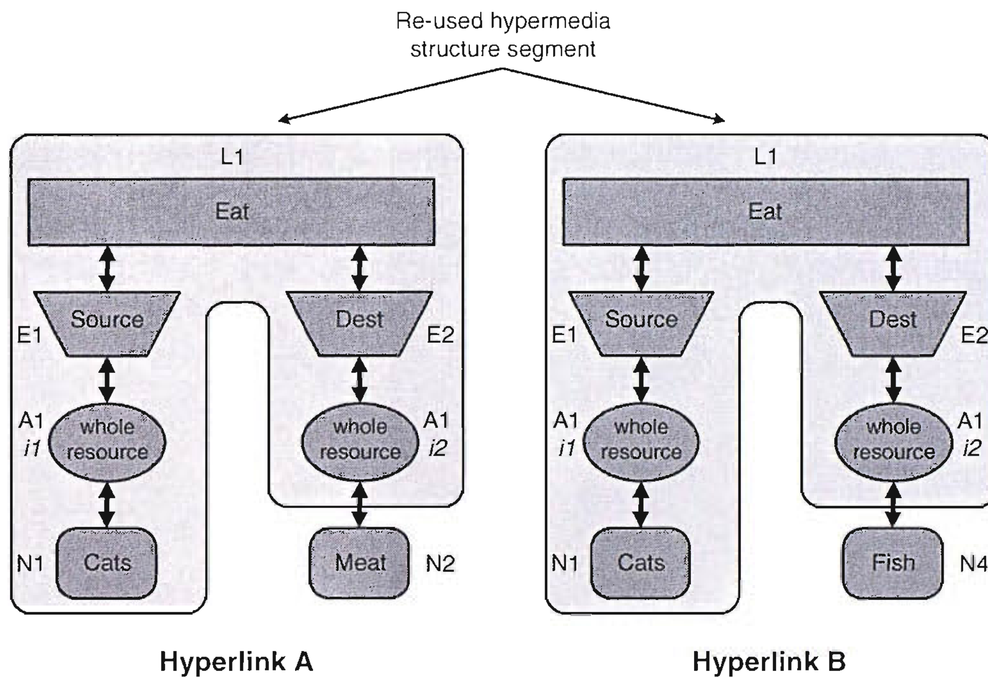


Figure B.4: Hyperlinks A and B of the 'Cats Eat' something relationship sharing the same structure segment as described by the OCS solution of Figure B.3.

B.3.2. Node-less 'Eat' Relationship

Section 6.3.1 also described how it would be useful to be able to create a re-usable Node-less 'Eat' relationship comprising objects A1, E1, L1, E2 and A2. This is not possible using OHP-Nav object reference embedding, but is possible using the OCS data model.

The OCS solution is shown in Figure B.5. This solution relies on the existing Function Objects within Function Object Space OS80 (Figure B.3) along with two new Function Objects shown in Function Object Space OS81. Figure B.5 also includes Connection Space CS81 which contains the appropriate Connection Objects to build the necessary OHP-Nav hypermedia structures. Table B.4 is included to underscore how and where the re-use is taking place within the OCS structures.

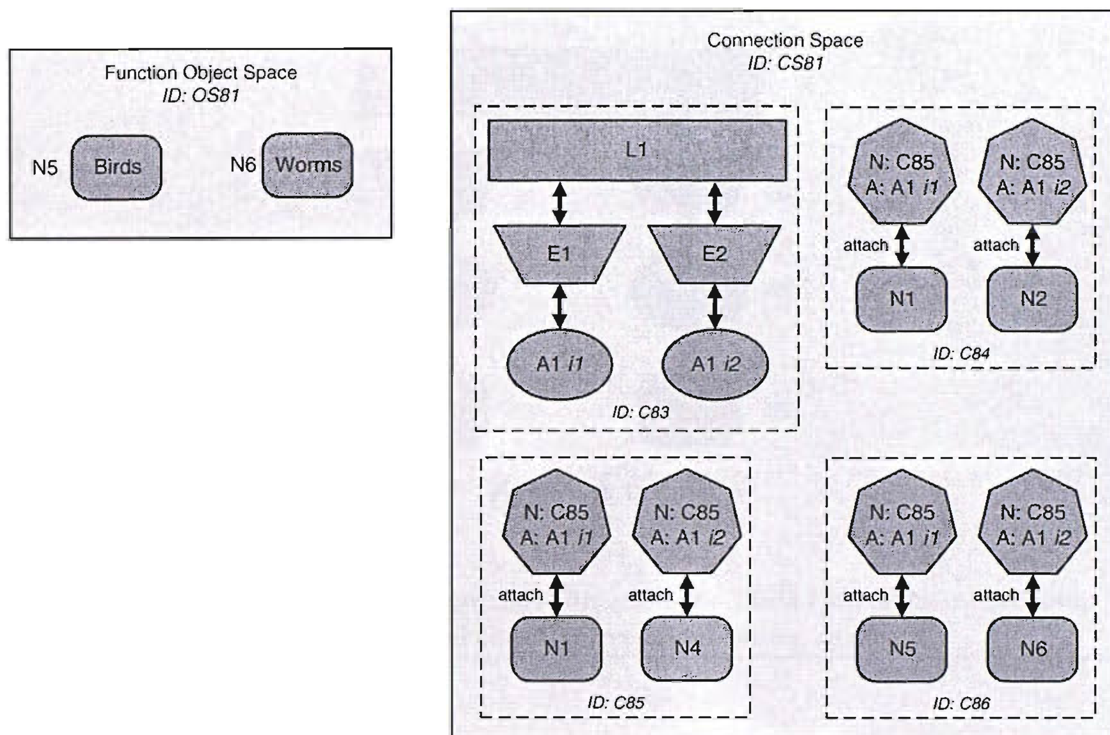


Figure B.5: OCS solution for the OHP-Nav Node-less relationship of Section 6.3.1. It also utilises Function Object Space OS80 of Figure B.3

The 'Eat' relationship is captured by Connection Object C83. It can be used to create an assortment of different 'Eat' relationships. Three examples are shown. The original 'Cats Eat Meat' and 'Cats Eat Fish' structures are captured by Connection Objects C84 and C85 respectively, and new relationship 'Birds Eat Worms' is captured by Connection Object C86.

Figure B.6 shows the latest relationship 'Birds Eat Worms' as it would be represented by the conventional OHP-Nav object reference embedding approach that prevents re-use of the 'Eat' relationship. This is to aid understanding Table B.4 which describes the hypermedia object re-use and the number of hypermedia objects saved as a result of adopting the OCS solution of Figure B.5.

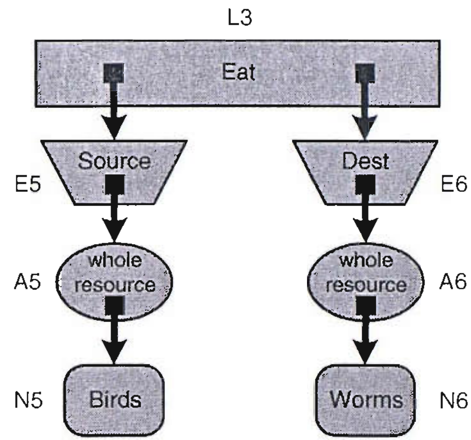


Figure B.6: OHP-Nav hypermedia structure depicting the 'Birds Eat Worms' relationship using the conventional object reference embedding approach.

Description	Value
No. of objects to build original OHP-Nav structures	21
No. of functionally identical objects	17
No. of objects to build OCS solution	13
No. of Function Objects	9
No. of Connection Objects	4
Re-used object count	5
Re-used object list	L1 for L2, L3 E1 for E3, E5 E2 for E4, E6 A1 for A2, A3, A4, A5, A6 N1 for N3
Eliminated object count	12
Eliminated object list	L2, L3, E3, E5, E4, E6, A2, A3, A4, A5, A6, N3
No. of overall objects saved	8
No. of equivalent hypermedia objects saved	12

Table B.4: Summary of the re-use taking place within the OCS solution of Figure B.5.

As indicated by Table B.4, this OCS solution uses a total of 13 objects. This compares with a total of 21 objects if using the OHP-Nav object reference embedding approach. Thus a total of 8 objects have been saved, and if the 4 Connection Objects are excluded (because they are deemed to be lightweight objects), then there is actually a saving of 12 equivalent hypermedia objects.

Figure B.7 shows the three hypermedia structures (each containing the same Node-less 'Eat' relationship) created as a result of instantiating C84, C85 and C86.

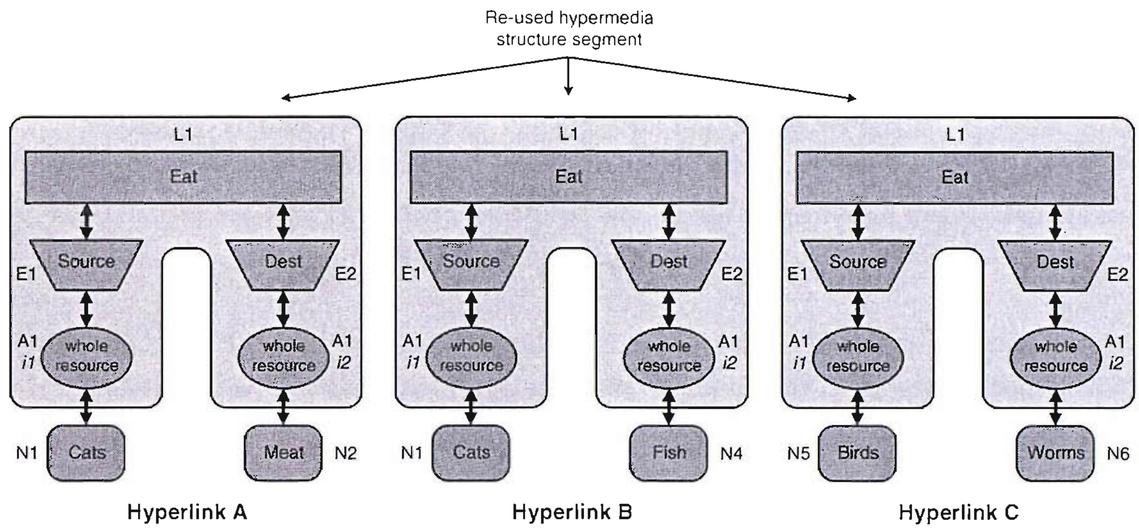


Figure B.7: Graphical depiction of the hypertext links of the OCS solution of Figure B.5.

References

Acksyn, Robert M., McCracken, Donald L. and Yoder, Elise A. 1988. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. *Communications of the ACM*, 31(7), 820-835.

Anderson, K., Taylor, R. and Whitehead, E. J. 1994. Chimera: Hypertext for Heterogeneous, Software Environments. *Pages 94-107 of: ECHT'94. Proceedings of the ACM European Conference on Hypermedia Technology, 18-23 September 1994, Edinburgh, UK.*

Anderson, Kenneth M., Sherba, Susanne A. and Van Lepthien, William. 2003a. Structural Templates and Transformations: The Themis Structural Computing Environment. *Journal of Network and Computer Applications*, 26(1), 47-71.

Anderson, Kenneth M., Sherba, Susanne A. and Van Lepthien, William. 2003b. Structure and Behavior Awareness in Themis. *Pages 138-147 of: Hypertext '03. Proceedings of the fourteenth ACM conference on Hypertext and Hypermedia, 26-30 August 2003, Nottingham, U.K.*

Ashman, Helen, Garrido, Alejandra and Oinas-Kukkonen, H. 1997. Hand-Made and Computed Links, Precomputed and Dynamic Links. *Pages 191-208 of: HIM '97. Proceedings of the Hypermedia - Information Retrieval - Multimedia, 1997, Germany.*

Ashman, Helen. 2000a. Electronic Document Addressing: Dealing with Change. *ACM Computing Surveys*, 32(3), 201-212.

Ashman, Helen. 2000b. Relations modelling sets of hypermedia links and navigation. *Computer Journal*, 43(5), 345-363.

Bailey, Christopher, Hall, Wendy, Millard, David E. and Weal, Mark J. 2002. A Structural Approach to Adaptive Hypermedia. *Proceedings of the 2nd International Conference on Adaptive Hypermedia and Adaptive Web Based Systems, May 2002, Malaga, Spain.*

Bendix, Lars, Dattolo, A. and Vitali, Fabio. 2001. Software Configuration Management in Software and Hypermedia Engineering: A Survey. *Handbook of Software Engineering and Knowledge Engineering*, Volume 1, 532-548.

Berners-Lee, Tim, Cailliau, Robert, Groff, Jean-Francois and Pollerman, Bernard. 1992. World Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 1(2), 52-58.

Berners-Lee, Tim. 1998. Semantic Web Road Map. *World Wide Web Consortium (W3C). Available as http://www.w3.org/DesignIssues/Semantic.*

Bernstein, M., J. Bolter, M., Joyce, M. and Mylonas, E. 1991. Architectures for volatile hypertext. *Pages 243-260 of: Hypertext '91. Proceedings of the third annual ACM conference on Hypertext, 15-18 December 1991, San Antonio, Texas, USA.*

Bernstein, M. 1996. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2), 86-98.

Bouvin, Niels Olof. 2000. Experiences with OHP and issues for the future. *Pages 13-22 of: OHS 6. Proceedings of the Sixth Workshop on Open Hypermedia Systems, ACM Hypertext 2000, 30 May - 3 June, 2003, San Antonio, Texas, USA.*

Brush, A. J. Bernheim. 2002. Annotating Digital Documents for Asynchronous Collaboration. *Technical Report for Department of Computer Science and Engineering*, University of Washington.

Bush, Vannevar. 1945. As We May Think. *Atlantic Monthly*, 176(1), 101-108.

Cailliau, Robert and Ashman, Helen. 1999. Hypertext in the Web - a History. *ACM Computing Surveys (electronic edition)*, 31(4).

Campbell, Brad and Goodman, Joseph M. 1988. HAM: A General Purpose Hypertext Abstract Machine. *Communications of the ACM*, 31(7), 856-861.

Carr, Les A., Hall, Wendy, Davis, Hugh C., De Roure, Dave C. and Hollom, R. 1994. The Microcosm Link Service and its Application to the World Wide Web. *Proceedings of the First World Wide Web Conference, 25-27 May 1994, Geneva, Switzerland.*

Conradi, Reidar and Westfechtel, Bernhard. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2), 232-282.

Creech, Michael L. 1996. Author-Oriented Link Management. *Computer Networks*, 28(7-11), 1015-1025.

Crimi, Alfred D. Sketch of memex, Life Magazine, 1945.

Davis, Hugh C., Hall, Wendy, Heath, Ian, Hill, Gary J. and Wilkins, R. J. 1992. Towards An Integrated Environment with Open Hypermedia Systems. *Pages 181-190 of: ECHT '92. Proceedings of the ACM conference on Hypertext, 30 November - 4 December 1992, Milan, Italy.*

Davis, Hugh C. 1995a. Data Integrity Problems in an Open Hypermedia Link Service. PhD Thesis, University of Southampton.

Davis, Hugh C. 1995b. To Embed or Not to Embed... *Communications of the ACM*, 38(6), 108-109.

Davis, Hugh C. 1996. *Proceedings of the 2.5 Open Hypermedia System Working Group Meeting, 7-8 December 1996, Southampton, UK.*

Davis, Hugh C., Lewis, Andy and Rizk, Antoine. 1996. OHP: A Draft Proposal for a Standard Open Hypermedia Protocol. *Pages 27-53 of: OHS 2. Proceedings of the Second Workshop on Open Hypermedia Systems, Hypertext '96, 16-20 March 1996, Washington DC, USA.*

Davis, Hugh C. and Hall, Wendy. 1997. The Microcosm Approach to Open Hypermedia (A Technical Overview). *Tutorial, Hypertext '97, 6-11 April 1997, Southampton, UK.*

Davis, Hugh C. 1998. Referential Integrity of Links in Open Hypermedia Systems. *Pages 207-216 of: Hypertext '98. Proceedings of the ninth ACM conference on Hypertext and Hypermedia, 20-24 June 1998, Pittsburgh, Pennsylvania, USA.*

Davis, Hugh C. 1999. Hypertext link integrity. *ACM Computing Surveys (electronic edition)*, 31(4).

Delisle, Norman M. and Schwarz, M. 1986. Neptune: A Hypertext System for CAD Applications. *Pages 132-143 of: Proceedings of the International Conference on Management of Data, 28-30 May 1986, Washington DC, USA.*

Engelbart, D. C. and English, William K. 1968. A Research Center for Augmenting Human Intellect. *Pages 396-410 of: AFIPS Conference. Proceedings of the 1968 Fall Joint Computer Conference, December 1968, San Francisco, CA, USA.*

Engelbart, Douglas C. 1998. Keynote Speech. *OHS 4. Proceedings of the fourth International Workshop on Open Hypermedia Systems, Hypertext '98, June 1998, Pittsburgh, USA.*

Fielding, Roy. 1994. Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web. *Proceedings of the First International World-Wide Web Conference, May 1994, Geneva, Switzerland.*

Flannagan, David. 1997. Javascript: The Definitive Guide. *O'Reilly & Associates.*

Griffiths, Jon-Paul, Reich, S. and Davis, Hugh C. 1999. The ContentSpec Protocol: providing Document Management Services for OHP. *Pages 29-33 of: OHS 5. Proceedings of the Fifth Workshop on Open Hypermedia Systems, Hypertext '99, February 1999, Darmstadt, Germany.*

Griffiths, Jon-Paul, Millard, David E., Davis, Hugh C., Michaelides, Danius T. and Weal, Mark J. 2002. Reconciling Versioning and Context in Hypermedia Structure Servers. *Pages 118-131 of: MIS 2002 (Metainformatics International Symposium 2002). Proceedings of the, 7-10 August 2002, Esbjerg, Denmark.*

Grønbæk, Kaj, Hem, J. A., Madsen, O. L. and Sloth, L. 1994. Cooperative Hypermedia Systems: A Dexter-based Architecture. *Communications of the ACM*, 37(2), 64-74.

Grønbæk, Kaj and Trigg, R. H. 1994. Design Issues for a Dexter-based Hypermedia System. *Communications of the ACM*, 37(2), 40-49.

Haake, Anja. 1992. CoVer: A Contextual Version Server for Hypertext Applications. *Pages 43-52 of: ECHT '92. Proceedings of the ACM conference on Hypertext, 30 November - 4 December 1992, Milan, Italy.*

Haake, Anja. 1994. Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications. *Pages 81-93 of: ECHT '94. Proceedings of the ACM European conference on Hypermedia technology, 19-23 September 1994, Edinburgh, Scotland, UK.*

Haake, Anja and Haake, Jörge M. 1993. Take CoVer: Exploiting Version Support in Cooperative Systems. *Pages 406-413 of: SIGCHI. Proceedings of the conference on Human factors in computing systems, 24-29 April 1993, Amsterdam, The Netherlands.*

Halasz, Frank and Schwarz, Mayer. 1990. The Dexter Hypertext Reference Model. *Pages 95-133 of: Proceedings of the Hypertext Workshop, National Institute of Standards and Technology, 16-18 January 1990, Gaithersburg, MD, USA.*

Halasz, Frank and Schwarz, Mayer. 1994. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2), 30-39.

Halasz, Frank G. 1988. Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, 31(7), 836-852.

Hicks, David L., Leggett, John J., Nürnberg, Peter J. and Schnase, J. L. 1998. Hypermedia Version Control Framework. *ACM Transactions on Information Systems*, 16(2), 127-160.

Ingham, D., Caughey, S. and Little, M. 1996. Fixing the "Broken-Link" Problem: The W3Object Approach. Pages 1255-1268 of: Proceedings of the Fifth International World-Wide Web Conference, May 1996, Paris, France.

Kim, Sunghun, Pan, Kai , Sinderson, Elias and Whitehead, E. James Jr. 2004. Architecture and Data Model of a WebDAV-based Collaborative System. Pages 48-55 of: CTS '04. Proceedings of the 2004 Collaborative Technologies Symposium, Western MultiConference, 18-21 January 2004, San Diego, California, USA.

Leggett, J. J. and Schnase, J. L. 1994. Viewing Dexter with Open Eyes. *Communications of the ACM*, 37(2), 77-86.

Leistøl, G. 1994. Aesthetic and rhetorical aspects of linking video in hypermedia. *Pages 217-223 of: ECHT '94. Proceedings of the ACM European Conference on Hypermedia Technology, 18-23 September 1994, Edinburgh, UK.*

Malcolm, Kathryn C., Poltrock, Steven E. and Schuler, Douglas. 1991. Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise. *Pages 13-24 of: Hypertext '91. Proceedings of the third annual ACM conference on Hypertext, 15-18 December 1991, San Antonio, Texas, USA.*

Marshall, C. C. and Shipman, Frank M. 1995. Spatial Hypertext: Designing for Change. *Communications of the ACM*, 38(8), 88-97.

Marshall, Catherine C. and Shipman, Frank M., III. 1997. Spatial Hypertext and the Practice of Information Triage. *Pages 124-133 of: Hypertext '96. Proceedings of the eighth ACM conference on Hypertext, 6-11 April 1997, Southampton, UK.*

McCall, R., Bennett, P., D'Ooronzio, P., Ostwald, J., Shipman, Frank M. and Wallace, N. 1990. PHIDIAS: Integrating CAD graphics into Dynamic Hypertext. *Hypertext: concepts, systems and applications*, 152-165.

McCracken, Donald L. and Acksyn, Robert M. 1984. Experience with the ZOG human-computer interface system. *International Journal of Man-Machine Studies*, 21(4), 293-310.

Meyrowitz, Norman K. 1989. The Missing Link: Why We're All Doing Hypertext Wrong. *The Society of Text: Hypertext, Hypermedia, and the Social Construction of Information*, 107-114.

Michaelides, D. T., Millard, D. E., Weal, M. J. and De Roure, D. 2001. Auld Leaky: A Contextual Open Hypermedia Link Server. *Pages 59-70 of: OHS 7. Proceedings of the Seventh Workshop on Open Hypermedia Systems, Hypertext 2001, Åarhus, Denmark.*

Microsoft. 2004. Dynamic Data Exchange. *MSDN Library, Microsoft. Available as http://msdn.microsoft.com/library/default.asp.*

Millard, David E., Chandler, David, Pind, L., Sloth, Lennert, Davis, Hugh C., Nürnberg, Peter J., Reich, S., Wiil, Uffe K., Grønbæk, Kaj, Anderson, Kenneth M. and Griffiths, Jon-Paul. 1998a. Talking OHP-Nav - Demonstrating Interoperability in Open Hypermedia Systems. *Hypertext '98 Demonstration. The ninth ACM conference on Hypertext and Hypermedia, 20-24 June 1998, Pittsburgh, USA.*

Millard, David E., Reich, S. and Davis, Hugh C. 1998b. Reworking OHP: the Road to OHP-Nav. *Pages 48-53 of: OHS 4. Proceedings of the Fourth International Workshop on Open Hypermedia Systems, Hypertext '98, Pittsburgh, USA.*

Millard, David E. 2000a. Hypermedia Interoperability: Navigating the Information Continuum. PhD Thesis, University of Southampton.

Millard, David E. and Davis, Hugh C. 2000. Navigating Spaces: The Semantics of Cross Domain Interoperability. *Pages 129-139 of: OHS 6. Proceedings of the Sixth Workshop on Open Hypermedia Systems, Hypertext 2000, 30 May - 3 June 2000, San Antonio, Texas, USA.*

Millard, David E., Davis, Hugh C. and Moreau, Luc. 2000a. Standardizing Hypertext: Where Next for OHP? *Pages 3-12 of: OHS 6 - The Sixth Workshop on Open*

Hypermedia Systems, Hypertext 2000. Proceedings of the, 30 May - 3 June 2000, San Antonio, Texas, USA.

Millard, David E., Moreau, Luc, Davis, Hugh C. and Reich, S. 2000b. FOHM: A Fundamental Open Hypertext Model for Investigating Interoperability between Hypertext Domains. *Pages 93-102 of: Hypertext 2000. Proceedings of the eleventh ACM on Hypertext and Hypermedia, 30 May - 3 June 2000, San Antonio, Texas, USA.*

Millard, David E. 2003. Discussions at the Data Border: From Generalised Hypertext to Structural Computing. *Journal of Network and Computer Applications*, 26(1), 95-114.

Nelson, T. H. 1981. Literary Machines. *Eastgate Systems Inc.*

Nelson, T. H. 1995. The Heart of Connection: Hypermedia Unified by Transclusion. *Communications of the ACM*, 38(8), 31-33.

Nelson, T. H. 1999a. Xanalogical Structure, Needed More Now Than Ever: Parallel Documents, Deep Links to Content, Deep Versioning, and Deep Re-Use. *ACM Computing Surveys (electronic edition)*, 31(4).

Nelson, T. H. 1999b. The Unfinished Revolution and Xanadu. *ACM Computing Surveys (electronic edition)*, 31(4).

Nelson, T. H. 1999c. The Xanadu Model: A Pictorial View of Referential Documents and Content Linking. *Floating World: Preliminary 1999 Specifications. Available as http://xanadu.com/zigzag/fw99/XUmodel.html.*

Nürnberg, Peter J., Leggett, J. J., Schneider, E. R. and Schnase, J. L. 1996. Hypermedia Operating Systems: A New Paradigm for Computing. *Pages 194-202 of: Hypertext '96. Proceedings of the seventh ACM conference on Hypertext, 16-20 March 1996, Washington DC, USA.*

Nürnberg, Peter J., Leggett, John J. and Schneider, Erich R. 1997. As We Should Have Thought. *Pages 96-101 of: Hypertext '97. Proceedings of the eighth ACM conference on Hypertext, 6-11 April 1997, Southampton, UK.*

Nürnberg, Peter J. 1999. Proceedings of the First Workshop on Structural Computing. *Pages of: First Workshop on Structural Computing, Hypertext'99. Proceedings of the, 21-25 February 1999, Darmstadt, Germany.*

Nürnberg, Peter J. 2002. Building Metainformatical Bridges. *Pages 6-8 of: MIS 2002. Proceedings of the Metainformatics International Symposium 2002, 7-10 August 2002, Esbjerg, Denmark.*

Nürnberg, Peter J. and Schraefel, Monica C. 2003. Relationships Among Structural Computing and other Fields. *Journal of Network and Computer Applications*, 26(11), 11-26.

OHSWG. 1997. OHSWG Compendium. *Available as http://www.csdl.tamu.edu/ohs/.*

OMG. 1998. The Common Object Request Broker: Architecture and Specification, Technical Report. Object Management Group (OMG), Revision 2.2.

Østerbye, Kasper. 1992. Structural and Cognitive Problems in Providing Version Control for Hypertext. *Pages 33-42 of: ECHT '92. Proceedings of the ACM conference on Hypertext 1992, 30 November - 4 December 1992, Milan, Italy.*

Pearl, Amy. 1989. Sun's Link Service: A Protocol for Open Linking. *Pages 137-146 of: Hypertext '89. Proceedings of the second annual ACM conference on Hypertext, 1989, Pittsburgh, PA, USA.*

Pitkow, James E. and Jones, R. Kipp. 1996. Supporting the Web: A Distributed Hyperlink Database System. *Pages 981 - 991 of: WWW5. Proceedings of the Fifth World Wide Web Conference, May 1996, Paris, France.*

Reich, S., Wiil, Uffe K., Nürnberg, Peter J., Anderson, Kenneth M., Millard, David E. and Haake, Jörge M. 1999a. Addressing Interoperability in Open Hypermedia: The Design of the Open Hypermedia Protocol. *New Review of Hypermedia and Multimedia*, Volume 5, 207-248.

Reich, S., Wiil, Uffe K., Nürnberg, Peter J., Anderson, Kenneth M., Millard, David E. and Haake, Jörge M. 2000. Addressing Interoperability in Open Hypermedia: The Design of the Open Hypermedia Protocol. *New Review of Hypermedia and Multimedia*, 207-243.

Reich, Siegfried, Griffiths, Jon-Paul, Millard, David E. and Davis, Hugh C. 1999b. Solent - A Platform for Distributed Hypermedia Applications. *Pages 802-811 of: Proceedings of the 10th International Conference on Database and Expert Systems Applications, 30 August - 3 September 1999, Florence, Italy.*

Reich, Siegfried and Millard, David E. 1999. OHP-Nav DTD (version 1.3 Darmstadt). *Available as http://www.ifs.uni-linz.ac.at/ifs/staff/reich/ohs/docs/defs.html.*

Reinert, O., Bucka-Lassen, D., Pedersen, C. A. and Nürnberg, Peter J. 1999. CAOS: A Collaborative and Open Spatial Structure Service Component with Incremental Spatial Parsing. *Pages 49-50 of: Hypertext '99. Proceedings of the tenth ACM Conference on Hypertext and Hypermedia, 21-25 February 1999, Darmstadt, Germany.*

Sawhney, N., Balcom, D. and Smith, I. 1996. HyperCafe: narrative and aesthetic properties of hypervideo. *Pages 1-10 of: Hypertext '96. Proceedings of the seventh ACM conference on Hypertext, 16-20 March 1996, Washington DC, USA.*

Soares, L. F. G., Rodrigues, N. L. R. and Casanova, M. A. 1993a. An Open Hypermedia System with Nested Composite Nodes and Version Control. *Technical Report*, Departamento de Informatica, PUC-Rio. Rio de Janeiro, Brasil.

Soares, L. F. G., Filho, G. L. d. S., Rodrigues, R. F. and Muchaluat, D. 1999. Versioning Support in the HyperProp System. *Multimedia Tools and Applications*, 8(3), 325-339.

Soares, L. F. G. , Casanova, M. A. and Colcher, S. 1993b. An architecture for hypermedia systems using MHEG standard objects interchange. *Information Services and Use*, 13(4), 131-139.

Soares, Luiz Fernando G. and Casanova, M. A. 1994. Nested Composite Nodes and Version Control in Hypermedia Systems. *Pages of: Proceedings of the Workshop on Versioning in Hypertext Systems, ECHT '94, September 1994, Edinburgh, UK.*

Sollins, Karen and Masinter, Larry. 1994. RFC 1737: Functional Requirements for Uniform Resource Names. *Internet Engineering Task Force (IETF). Available as http://www.ietf.org/rfc/rfc1737.txt.*

Tichy, W. F. 1985. RCS - A System for Version Control. *Software-Practice and Experience*, 15(7), 637-654.

Tzagarakis, Manolis, Avramidis, Dimitris, Kyriakopoulou, Maria, Schraefel, Monica C., Viaitis, Michalis and Christophodoulakis, Dimitris. 2003. Structuring Primitives in the Callimachus Component-Based Open Hypermedia System. *Journal of Network and Computer Applications*, 26(1), 139-162.

Vanzyl, Adrian J., Cesnik, Branko , Heath, Ian and Davis, Hugh C. 1994. Open Hypertext Systems: an Examination of Requirements, and Analysis of

Implementation Strategies, Comparing Microcosm, HyperTED and the WWW. *Available as http://www.infwiss.uni-konstanz.de/Res/openhypermedia.html.*

Vitali, Fabio. 1999. Versioning Hypermedia. *ACM Computing Surveys (electronic edition)*, 31(4).

Weal, Mark J., Millard, David E., Michaelides, Danius T. and De Roure, David. 2001. Building Narrative Structures Using Context Based Linking. *Pages 37-38 of: Hypertext '01. Proceedings of the twelfth ACM conference on Hypertext and Hypermedia, 14-18 August 2001, Åarhus, Denmark.*

Whitehead, E. James Jr., Anderson, Kenneth M. and Taylor, R. N. 1994. A Proposal for Versioning Support for the Chimera System. *Pages 45-54 of: Proceedings of the Workshop on Versioning in Hypertext Systems, ECHT '94, Edinburgh, UK.*

Whitehead, E. James, Jr. 2001a. WebDAV and DeltaV: Collaborative Authoring, Versioning, and Configuration Management for the Web. *Pages 259-260 of: Hypertext '01. Proceedings of the twelfth ACM conference on Hypertext and Hypermedia, 14-18 August 2001, Åarhus, Denmark.*

Whitehead, E. James, Jr. 2001b. Design Spaces for Link and Structure Versioning. *Pages 195-204 of: Hypertext '01. Proceedings of the twelfth ACM conference on Hypertext and Hypermedia, 14-18 August 2001, Åarhus, Denmark.*

Whitehead, E. James Jr. and Goland, Yaron Y. 2004. The WebDAV Property Design. *Software - Practice and Experience*, 34(2), 135-161.

Wiil, Uffe K. 1993. Experiences with HyperBase: A Hypertext Database Supporting Collaborative Work. *SIGMOD Record*, 22(4), 19-25.

Wiil, Uffe K. and Leggett, John J. 1996. The HyperDisco Approach to Open Hypermedia Systems. *Pages 140-148 of: Hypertext '96. Proceedings of the seventh ACM conference on Hypertext, 16-20 March 1996, Washington DC, USA.*

Wiil, Uffe K. and Leggett, J. J. 1997. Workspaces: The HyperDisco Approach To Internet Distribution. *Pages 13-23 of: Hypertext '97. Proceedings of the eighth ACM conference on Hypertext, April 6-11, Southampton, UK.*

Wiil, Uffe K. 1998. Open Hypermedia: Systems, Interoperability and Standards. *Journal of Digital information*, 1(2).

Wiil, Uffe K. 1999. Multiple Open Services in a Structural Computing Environment. *Pages 34-39 of: Proceedings of the First Workshop on Structural Computing, Hypertext '99, 21-25 February 1999, Darmstadt, Germany.*

Wiil, Uffe K. and Nürnberg, P. J. 1999. Evolving Hypermedia Middleware Services: Lessons and Observations. *Pages 427-436 of: Proceedings of the Symposium on Applied Computing, 28 February - 2 March 1999, San Antonio, Texas.*

Wiil, Uffe K. 2000a. Using the Construct Development Environment to Generate a File-Based Hypermedia Storage Service. *Pages 147-159 of: Proceedings of the Second International Workshop on Structural Computing, Hypertext 2000, 30 May - 3 June 2000, San Antonio, Texas, USA.*

Wiil, Uffe K. 2000b. Towards a Proposal for a Standard Component-Based Open Hypermedia System Storage Interface. *Pages 23-30 of: OHS 6. Proceedings of the Sixth Workshop on Open Hypermedia Systems, Hypertext 2000, 30 May - 3 June, 2003, San Antonio, Texas, USA.*

Wiil, Uffe K., Nürnberg, Peter J., Hicks, David L. and Reich, Siegfried. 2000. A Development Environment for Building Component-Based Open Hypermedia Systems. *Pages 266-267 of: Hypertext 2000. Proceedings of the eleventh ACM on Hypertext and Hypermedia, 30 May - 3 June 2000, San Antonio, Texas, USA.*

Wiil, Uffe K. 2001. Development Tools in Component-Based Structural Computing Environments. *Pages 82-93 of: OHS 7. Proceedings of the Seventh Workshop on Open Hypermedia Systems, Hypertext '01, 14-18 August 2001, Århus, Denmark.*

Wiil, Uffe K. and Leggett, John J. 1992. Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems. *Pages 251-261 of: ECHT '92. Proceedings of the ACM conference on Hypertext 1992, 30 November - 4 December 1992, Milan, Italy.*

Wiil, Uffe K. , Hicks, David L. and Nürnberg, Peter J. 2001. Multiple Open Services: A New Approach to Service Provision in Open Hypermedia Systems. *Pages 83-92 of: Hypertext '01. Proceedings of the twelfth ACM conference on Hypertext and Hypermedia, 14-18 August 2001, Århus, Denmark.*

World RPS Society, The. 2002. The Official Rock Paper Scissors Strategy Guide. *Available as http://www.worldrps.com/.*

Yankelovich, N., Haan, B. J., Meyrowitz, N. K. and Drucker, S. M. 1988. Intermedia: The Concept and the Construction of a Seamless information Environment. *IEEE Computer*, 21(1), 81-83, 90-96.

Zhang, Li, Bieber, Michael, Millard, David E. and Oria, Vincent. 2004. Supporting Virtual Documents in Just-in-Time Hypermedia Systems. To appear in the *ACM Symposium on Document Engineering 2004, 28-30 October 2004, Milwaukee, Wisconsin, USA.*