Exploratory Programming of Distributed Augmented Environments

Blair MacIntyre

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1999

Abstract

Exploratory Programming of Distributed Augmented Environments

Blair MacIntyre

Augmented reality is a form of virtual reality that uses see-through displays to enhance the world with computer-generated material. When combined with more traditional displays, a powerful *augmented environment* emerges in which two and three dimensional information can be presented to a user simultaneously on a combination of displays. Prototyping these environments is challenging both because they are highly distributed, interactive systems, and because of the exploratory nature of building systems for new interaction paradigms.

We have developed a testbed for exploratory programming of distributed augmented environments, called Coterie. A single programming model is used for both single and multi-process programs by building applications as groups of threads communicating via shared objects. The distributed programming model is *distributed object memory* (DOM), an object-based approach to *distributed shared memory.* Coterie's DOM presents the programmer with both client-server and replicated distributed objects.

Both interpreted (Repo) and compiled (Modula-3) languages present the application programmer with similar DOM programming models. Modula-3's replicated objects are implemented using Shared Objects, an object replication package that is tightly integrated with the Modula-3 object system and designed to be flexible and easy-to-use. Repo is implemented using the Shared Object package, and presents the programmer with an interpreted language that supports both client-server and replicated objects uniformly across its entire type system. The final important component of Coterie is Repo-3D, a high-level, distributed graphics library, built using the Shared Objects package and tightly integrated with Repo. By making all graphical objects extensible and transparently distributable, programmers can use Repo-3D scene graphs as the basis for their application data

structures, allowing complex distributed graphical applications to be created in a straightforward manner.

Numerous stand-alone and distributed augmented environment systems have been developed using Coterie, and demonstrate its usefulness. These include an architectural anatomy system for viewing the support structures inside the walls of a building, a construction assistant for space frame buildings, a maintenance and repair task for telephone crossboxes, an augmented reality tour guide, and a number of interface concepts for the National Tele-Immersion Initiative.

Table of Contents

CHAPTER 1 Introduction 1

1.1 Exploratory Programming of Distributed Augmented Environments 3
1.2 Research Contributions 6
1.2.1 Shared Objects: A Distributed Shared Object Memory. 6
1.2.2 Repo: A Distributed Interpreted Language 8
1.2.3 Repo-3D: A Distributed 3D Graphics Library. 9
1.2.4 Coterie: Exploratory Programming of AE Systems. 11
1.2.5 Prototype Augmented Reality Applications 11

CHAPTER 2 An Overview of Coterie 13

2.1 Previous Work: Augmented Reality 13
2.1.1 KARMA. 14
2.1.2 Windows on the World. 15
2.1.3 Architectural Anatomy. 16
2.2 Motivation 17
2.3 Requirements for the Testbed 19
2.4 Related Research Areas 22
2.4.1 Virtual Environment Systems 22
2.4.2 Distributed Groupware. 25
2.5 Implementation Overview 26
2.5.1 Virtual Environments: Tracker Support 28
2.6 Initial Prototypes 30
2.6.1 Architectural Anatomy. 30
2.6.2 Telephone Crossbox Maintenance 31
2.6.3 Spaceframe Construction 32
2.6.4 Automated Tour Guide 33

CHAPTER 3 Shared Objects. 38

3.1 Distributed Shared Memory. 39
3.2 Related Work 42
3.3 Shared Object Package Design 47
3.3.1 Goal: Tight Integration. 47
3.3.2 Model: Totally Ordered, Write-Update Objects 48
3.3.3 Event Driven Control Flow: Callback Objects 53
3.4 Implementation 54
3.4.1 Object Definition and Runtime Code Generation 55

3.4.1.1 Example Object Definition . . . 56
3.4.1.2 Callback Object Usage . . . 61
3.4.1.3 Passing State Between Processes . . . 63
3.4.1.4 Additional Tracker Examples . . . 65
3.4.2 The Shared Object Runtime . . . 66
3.4.2.1 Thread Management . . . 68
3.4.2.2 Exception and Return Value Handling . . . 71
3.4.3 Restrictions . . . 72
3.5 Performance and Usability . . . 73
3.5.1 Shared Object Performance . . . 73
3.5.2 Shared Object Usability . . . 79
3.6 Discussion . . . 82

CHAPTER 4 Repo . . . 84

4.1 Related Work . . . 86
4.2 An Overview of Obliq and Repo . . . 87
4.3 Distributed Semantics . . . 88
4.4 Replication Syntax . . . 92
4.4.1 Declarations . . . 92
4.4.2 Cloning Data . . . 95
4.4.3 User-defined Picklers . . . 97
4.5 The Replication Module . . . 98
4.6 Examples . . . 99
4.6.1 Simple Tracker Report Distribution . . . 99
4.6.2 Asynchronous Change Notification . . . 101
4.6.3 Multi-person Spaceframe Construction . . . 102
4.6.4 Distributed Mutexes . . . 105
4.6.5 Hierarchical Object Directories . . . 108
4.7 Implementation . . . 111
4.8 Usability of Repo . . . 115

CHAPTER 5 Repo-3D . . . 118

5.1 Related Work . . . 121
5.2 Obliq-3D: An Overview . . . 123
5.3 Design Of Repo-3D . . . 126
5.3.1 Conversion to Shared Objects . . . 126
5.3.1.1 Graphical Objects . . . 127
5.3.1.2 Properties . . . 128
5.3.1.3 Animation Handles . . . 129
5.3.1.4 Input Callbacks . . . 131
5.3.1.5 Change Notification . . . 131

5.3.2 Local Variations . . . 133
5.3.3 Extensibility . . . 135
5.4 Examples . . . 137
5.4.1 A Tutorial Example . . . 138
5.4.2 Yet Another Tracker Example . . . 138
5.4.3 A Truncated Pyramid Object . . . 140
5.4.4 An Animation Examiner . . . 141
5.5 Implementation . . . 144
5.6 Performance . . . 153
5.7 Discussion . . . 155

CHAPTER 6 Coterie Examples. . . . 158

6.1 Of Vampire Mirrors and Privacy Lamps . . . 158
6.2 Shared Sketch. . . . 162

CHAPTER 7 Conclusions and Future Work . . . 170

7.1 Future Work . . . 174
7.1.1 Shared Object Update Latency . . . 175
7.1.2 Network Awareness . . . 176
7.1.3 Additional Replication Semantics . . . 178
7.1.4 Multi-object Consistency . . . 180
7.1.5 More Flexible Consistency Guarantees . . . 180
7.1.6 Better Handling of Time . . . 181
7.1.7 Generalized Local Variations in Repo-3D . . . 182
7.1.8 Application to Other Languages . . . 182

References . . . 184

APPENDIX A Example Generated Code . . . 190

A.1 TrackerPositionSO.m3 . . . 190
A.2 TrackerPositionCB.i3 . . . 197
A.3 TrackerPositionCB.m3 . . . 197
A.4 TrackerPositionProxy.i3 . . . 200
A.5 TrackerPositionCBProxy.i3 . . . 200
A.6 TrackerPositionPickle.i3 . . . 201

APPENDIX B Tracker Modules . . . 202

B.1 The Basic Modules . . . 202

B.1.1 Kalman . . . 202
B.1.2 Tracker . . . 202
B.1.3 TrackerPosition . . . 203
B.1.4 TrackerPositionCB . . . 203
B.1.5 TrackerServer . . . 204
B.2 The Tracking Device Modules . . . 204
B.2.1 Dynasight . . . 204
B.2.2 FOB . . . 205
B.2.3 Logitech . . . 205
B.2.4 MSMouse . . . 206
B.2.5 PTU . . . 206
B.2.6 RingMouse . . . 206
B.2.7 Scanner . . . 207
B.2.8 Trimble . . . 207
B.2.9 vIO . . . 208

APPENDIX C Repo Syntax . . . 209

APPENDIX D Additional Enhancements to Repo . . . 211

D.1 Additional Syntax Changes . . . 211
D.2 Module Enhancements and Additions . . . 213
D.3 Efficient Module Distribution . . . 215

APPENDIX E Repo Modules . . . 219

E.1 New Modules . . . 219
E.1.1 debug . . . 219
E.1.2 dict . . . 220
E.1.3 reflect . . . 220
E.1.4 replica . . . 222
E.2 New Modules for Modula-3 Packages . . . 223
E.2.1 dir . . . 223
E.2.2 http . . . 224
E.2.3 httpField . . . 229
E.2.4 httpStatus . . . 230
E.2.5 path . . . 231
E.2.6 random . . . 232
E.2.7 regex . . . 232
E.2.8 tcp . . . 233
E.2.9 url . . . 234
E.2.10 word . . . 235

E.3 Changed Modules 235
E.3.1 array 235
E.3.2 fmt 236
E.3.3 lex 236
E.3.4 net 237
E.3.5 os 238
E.3.6 process 238
E.3.7 sys 239
E.3.8 text 240
E.3.9 thread 241
E.4 Unchanged Modules 242
E.4.1 bool 243
E.4.2 char 243
E.4.3 color 243
E.4.4 form 244
E.4.5 int 245
E.4.6 math 246
E.4.7 online 246
E.4.8 pickle 247
E.4.9 rd 247
E.4.10 real 248
E.4.11 vbt 249
E.4.12 wr 249

APPENDIX F Another Replicated Mutex 251

F.1 mutex.obl 252

APPENDIX G Additional Enhancements To Repo-3D 254

APPENDIX H Repo-3D Modules 257

H.1 Graphics Objects 257
H.1.1 GO 257
H.1.2 GOCB 258
H.1.3 AmbientLightGO 260
H.1.4 BoxGO 260
H.1.5 CameraGO 260
H.1.6 ChoiceGroupGO 261
H.1.7 ConeGO 261
H.1.8 CylinderGO 262
H.1.9 DiskGO 262
H.1.10 GroupGO 263

H.1.11 IndexedLineSetGO . . . 263
H.1.12 IndexedPolygonSetGO . . . 264
H.1.13 LightGO . . . 264
H.1.14 LineGO . . . 265
H.1.15 MarkerGO . . . 265
H.1.16 OrthoCameraGO . . . 266
H.1.17 PerspCameraGO . . . 266
H.1.18 PointLightGO . . . 266
H.1.19 PolygonGO . . . 267
H.1.20 QuadMeshGO . . . 267
H.1.21 RootGO . . . 267
H.1.22 SphereGO . . . 268
H.1.23 SpotLightGO . . . 269
H.1.24 SurfaceGO . . . 269
H.1.25 Text2DGO . . . 271
H.1.26 TextGO . . . 271
H.1.27 TorusGO . . . 272
H.1.28 VectorLightGO . . . 272
H.2 Properties . . . 272
H.2.1 Prop . . . 272
H.2.2 PropCB . . . 273
H.2.3 BooleanProp . . . 273
H.2.4 ColorProp . . . 274
H.2.5 FontStyleProp . . . 274
H.2.6 IntProp . . . 275
H.2.7 LineTypeProp . . . 276
H.2.8 MarkerTypeProp . . . 276
H.2.9 Point2Prop . . . 277
H.2.10 PointProp . . . 278
H.2.11 RasterModeProp . . . 278
H.2.12 RealProp . . . 279
H.2.13 ShadingProp . . . 280
H.2.14 StringProp . . . 280
H.2.15 TexImageProp . . . 281
H.2.16 TexModelProp . . . 282
H.2.17 TextAlignProp . . . 282
H.2.18 TransformProp . . . 283
H.2.19 TransmissionPatternProp . . . 284
H.3 Animation Handles . . . 285
H.3.1 AnimHandle . . . 285
H.3.2 AnimHandleCB . . . 285
H.4 Interaction Callbacks . . . 286
H.4.1 KeyCB . . . 286

H.4.2 MouseCB 286
H.4.3 PositionCB 286
H.5 Location Callbacks 287
H.5.1 ProjectionCB 287
H.5.2 TransformCB 287
H.6 Graphics Bases 287
H.6.1 GraphicsBase 288
H.6.2 Win_OpenGL_Base 288
H.6.3 Win_RW_Base 288
H.6.4 X_OpenGL_Base 288
H.7 Miscellaneous 289
H.7.1 Anim3D 289
H.7.2 AnimHook 289
H.7.3 ProxiedObj 289
H.7.4 TessSphere 290
H.7.5 TexImage 290

APPENDIX I The Animation Time Module 291

I.1 animtime.obl 292

List of Figures

Figure 2-1: The KARMA prototype 14
Figure 2-2: The Windows on the World prototype 15
Figure 2-3: The Architectural Anatomy prototype 17
Figure 2-4: An example architecture diagram 18
Figure 2-5: The generic Tracker Report Object hierarchy 29
Figure 2-6: The new Architectural Anatomy prototype 31
Figure 2-7: A prototype AR application for crossbox maintenance 32
Figure 2-8: A prototype AR application for space frame construction 33
Figure 2-9: A prototype campus information system 34
Figure 2-10: Additional images of the Touring machine 35
Figure 2-11: Software design of the prototype campus information system 36
Figure 3-1: Control and data flow for a Shared Object update 52
Figure 3-2: The relationship between clients, sequencers and object managers 53
Figure 3-3: Object hierarchy for a Shared Object 54
Figure 3-4: The Modula-3 interface definition for `TrackerPosition` 57
Figure 3-5: The Modula-3 implementation for `TrackerPosition` 58
Figure 3-6: The `TrackerPositionCB.T` Callback Object 62
Figure 3-7: The default `TrackerPosition.T` marshalling code 64
Figure 3-8: A low frequency tracker object 66
Figure 3-9: Data Flow in the Shared Object System 69
Figure 4-1: The effect of different distribution semantics 90
Figure 4-2: Declaring objects in Repo 94
Figure 4-3: An example of synchronized replicated objects in Repo 100
Figure 4-4: An example of notifier callback objects in Repo 102
Figure 4-5: Extending the space frame prototype for remote consultation 103
Figure 4-6: The replicated state for the distributed ARC prototype 104
Figure 4-7: A simple client-server mutex 106
Figure 4-8: A simple replicated mutex 107
Figure 4-9: A single Object Directory (OD) 109
Figure 4-10: The internal definition of an Obliq array 112
Figure 4-11: The internal definition of a Repo array 113
Figure 4-12: The internal definition of a Repo replicated object 114
Figure 5-1: Two meanings of distributed graphics 119
Figure 5-2: The Repo-3D GO class hierarchy 124
Figure 5-3: The relationship between properties, names, values, and behaviors 125
Figure 5-4: The `GOCB` and `PropCB` modules 132
Figure 5-5: Embedding DistAnim-3D objects in Repo 136

Figure 5-6: A simple Repo-3D example 139
Figure 5-7: The `TruncPyr` object 141
Figure 5-8: The distributed CATHI animation viewer 143
Figure 5-9: The internal structure of Anim-3D and DistAnim-3D 145
Figure 5-10: The `GO.T` class 148
Figure 5-11: Excerpts from `GOPrivate.i3` 149
Figure 5-12: `BoxGO.T` class definitions 149
Figure 5-13: The `GroupGO.T` class definition 150
Figure 5-14: The `AnimHandle` class 152
Figure 6-1: The EMMIE system for collaborative augmented environments 159
Figure 6-2: The routine to create a VUB item 160
Figure 6-3: The structure of a VUB item's GO 161
Figure 6-4: The definition of a Coterie Sketch object 166
Figure 6-5: Distributed Sketch in use 167
Figure 6-6: The structure of the Distributed Sketch prototype 168
Figure D-1: Pattern matching with the Repo reflection module 214

List of Tables

Table 3-1: A comparison of distributed object-based programming systems 43
Table 3-2: Local method call performance 73
Table 3-3: Distributed method call performance. 74
Table 3-4: Orca Method call performance 75
Table 4-1: Entities with state in Obliq. 92
Table 4-2: Declaring entities with state in Repo 93
Table 6-1: Sketch Object Definitions 163

Acknowledgments

While it will be impossible to thank all those who have contributed in some way to this work, there are certain folks who must be acknowledged. First and foremost are the members of the Computer Graphics and User Interfaces Lab at Columbia University, where this work was undertaken, especially my advisor Steven Feiner. Xinshi Sha was instrumental in implementing many of the efficiency improvements to Anim3D to make it usable for our applications. Numerous others at Columbia influenced this work over the years, especially Clifford Beshers, Reza Jalili, Sushil Dasilva, Tobias Höllerer, Steven Dossick, Steven Abrams, Bruce Zenel, Andreas Butz and Simon Baker. Luca Cardelli and Marc Najork of DEC SRC created Obliq and Obliq-3D, and provided ongoing help and encouragement over the years that Repo and Repo-3D have been evolving. Bill Kalsow and Farshad Nayari of Critical Mass provided help with all aspects of Modula-3, including creating an excellent implementation of the language.

When examining my underlying motivations for how I approached this problem, the influence of Peter Buhr can be found, to whom I am thankful for long ago instilling in me the value of a solid and beautiful foundation for any complex programming system. I am grateful to Henri Bal for creating Orca and publishing inspiring and lucid papers on both the language and Distributed Object Memory in general.

Finally, I would like to thank the many relatives and friends who have encouraged and supported me over the many years spent on this work. My family has never wavered in their support, or in their conviction that I could and would one day complete this dissertation. Many, many friends, including those mentioned above, have provided the support needed over the years, sharing more than just food, coffee, beer and scotch. The most important of these is Beth Mynatt, whose love and encouragement kept me sane and focused during the final months of writing.

This work was supported by ONR Contracts N00014-94-1-0564 and N00014-97-1-0838, the National Science Foundation under Grant CDA-92-23009 and ECD-88-11111, the New York State Center for High Performance Computing and Communications in Healthcare (supported by the New York State Science and Technology Foundation), the Advanced Network & Services National Tele-Immersion Initiative, and gifts from NYNEX Science & Technology, Intel, Critical Mass, Apple, Microsoft, and Mitsubishi Electric Research Laboratory.

I also record those events which led, by insensible steps, to my after tale of misery, for when I would account to myself for the birth of that passion which afterwards ruled my destiny I find it arise, like a mountain river, from ignoble and almost forgotten sources; but, swelling as it proceeded, it became the torrent which, in its course, has swept away all my hopes and joys.

— Mary Shelley, *Frankenstein*

CHAPTER 1 Introduction

Are you sitting comfortably? Then I'll begin.

— Preamble to children's story in *Listen With Mother*, BBC radio program from 1950.

In recent years, the popularity of *virtual environments* (VEs), also known by the popular term *virtual reality* (VR), has varied greatly. While VEs initially received significant attention as an interface metaphor that would revolutionize the way people interact with computers, most attempts to explore this new paradigm have not lived up to initial expectations. The focus of much of this attention has been on what we shall refer to as *exclusive VEs*, in which an entirely synthetic sensory experience is presented to the user, typically focusing on the visual and auditory senses through the use of opaque, head-worn displays and headphones. The intuitive appeal of VEs is obvious; by immersing users in a rich 3D world, they should be able to interact with virtual information using the same skills with which they interact with the real world every day. However, there are a few obvious problems with the paradigm. The first is a technological one: even with the rapid advance in technology, it does not seem likely that convincing, realistic environments will be created any time soon, either in terms of the quality of the illusion presented to the user, or the quality of the interaction. The second is a social one: even if reasonably realistic virtual environments could be created, it is not clear that people will want their interactions with the "virtual world" to cut them off from the richness of the real world and their interactions with other people and physical artifacts, such as telephones, books, pens and paper.

Fortunately, alternative approaches may provide many of the proposed advantages of VR while avoiding many of the problems. In particular, *augmented reality* (AR) techniques aim to enhance, rather than replace, the user's perception of the world with com-

puter-generated information. Our view, shared by many AR researchers, is that exclusive VEs should (and probably will) only be used for those applications where the extreme sense of presence obtained by being entirely immersed in an virtual environment outweighs the disadvantages of being cut off from the real world. We also believe that only a small subset of the applications that have been proposed for VEs fit in this category. Furthermore, it is not clear that either pure AR or VR will completely replace existing computer interfaces. Instead, it seems likely that hybrid user-interfaces will emerge that combine AR with other interaction paradigms. These interfaces would integrate mainstream input and output devices (e.g, mice, tablets, speech generation, voice recognition, and desktop, wall-mounted and handheld displays) with techniques and devices from such interaction paradigms as augmented reality, ubiquitous computing and intelligent environments. Such interfaces are called *augmented environment*s (AEs).

The importance of AR techniques in these environments lie in their *personal* nature. Since AR displays tend to be *personal displays* (that are worn or carried by an individual user and whose output is only perceived by that user), AR techniques provide a solid foundation for imbuing an augmented environment with ubiquitous display capabilities that are tailored to the needs of individual users. By accurately sensing the location and orientation of the occupants of the environment, auditory and visual augmentations can also be presented *in context* on a per user basis. When combined with more traditional devices, a rich information space is created where combinations of public and private information can be presented to users simultaneously using a combination of displays. This augmenting and leveraging of the real world makes possible a wide variety of interaction techniques and ways of organizing information.

The presence of these personal AR displays makes AEs especially complex, for three reasons: the displays are available *continuously*, the information environment changes *dynamically*, and the information often needs to be *spatially located*. In the first case, since the displays are designed to be perceived *at the same time* that the user is attending to naturally occurring phenomenon with the same senses used to perceive the AR display, the information being presented must be designed so that it does not interfere with their day-to-day activities. Second, even when the user is attending to the information

presented on the see-through head-worn displays or headphones, the information continues to interact with the environment, in possibly complex ways. For example, small movements of the user's head will change what they are looking at and may therefore require information on the (personal or shared) displays to be changed. These interactions must then be taken into account when designing these dynamically changing information displays. Finally, virtual information often needs to appear directly in some physical context, either by overlaying visual elements at some specific location, or by spatially positioning a sound. To do this, highly accurate position and orientation sensing of the people and objects that occupy the environments is needed.

1.1 Exploratory Programming of Distributed Augmented Environments

The research reported in this dissertation was motivated by our interest in, and investigation of, the user-interaction issues that arise when building applications for multi-user AEs [MacIntyre and Feiner, 1994, MacIntyre and Feiner, 1996a, MacIntyre and Mynatt, 1998]. Based on our initial experience building single-user AR prototypes [Feiner et al., 1993a, Feiner et al., 1993b, Feiner et al., 1995], it was obvious early on that exploring multi-user AEs would be unusually challenging. First, the physical environments are extremely difficult to work with. Multiple users, multiple displays of different kinds (from see-through head-worn to wall-mounted to hand-held), and a wide variety of input devices (from pens and mice to voice to three and six degree-of-freedom (DOF) sensors) must be integrated into a single cohesive system. Second, these devices and displays are attached to an assortment of computers, requiring that the simplest of applications be distributed over many machines. These difficulties are exacerbated by the *exploratory* nature of building prototypes for a completely new interaction paradigm: neither the structure of the applications, the kind of data being shared, nor the distribution characteristics of that data are necessarily known ahead of time and will likely be modified continuously as the applications are developed.

Finally, our research prototypes are usually developed by a group of researchers, not an individual programmer. Dourish points out that not only the mechanics, but also the

social activities, of prototyping and collaborative development require toolkits with a high degree of flexibility, a finding that falls in line with our experiences. If the tools are too rigid, the natural exploratory nature of collaborative prototyping suffers [Dourish, 1996].

This research meets these challenges by combining state-of-the-art distributed systems, programming language and graphics techniques into a flexible programming environment called Coterie. In contrast to much of the research in these areas, the issues encountered when building Coterie are addressed in the context of our main research focus, the exploration of novel, highly interactive user-interfaces. As a result, Coterie is well suited to prototyping applications for multiple display, multi-modal augmented environments, as well as a wide variety of other distributed interactive applications.

The goal of this research is to create an environment that allows distributed programs to be built as easily as non-distributed ones, even if that means the execution and network usage of those programs may be slightly less efficient than otherwise possible. The primary concern is to develop an exploratory programming environment in which (in the words of Alan Kay) “simple things should be simple, complex things should be possible”. This focus on ease of use, at the possible expense of efficiency, is not typical in the design of distributed programming environments. However, in the context of our motivating research agenda, this focus is reasonable: the target audience is HCI researchers, ranging from undergraduate students to graduate students, professors and professional programmers, most of whom have little experience (or direct interest in) building complex distributed applications. These researchers typically build throw-away application prototypes to demonstrate interaction concepts, and are not overly focused on the efficiency of execution of these prototypes.

We satisfy these goals by focusing on *simplicity* and *flexibility.* To ensure that “simple things are simple”, Coterie’s distributed programming model is based on a familiar and well understood non-distributed programming paradigm, that of multiple threads of control communicating via shared objects. By providing an object-based implementation of *distributed shared memory* (DSM) [Li, 1986], often called a *distributed object memory* (DOM) [Levelt et al., 1992], both stand-alone and distributed programs are built the same way, with local and distributed data being used transparently and inter-

changeably, and with threads on the same or different machines communicating through shared objects. Furthermore, these distribution facilities are tightly-integrated with a popular programming language, allowing existing software and programming skills to be capitalized on. Our contribution to Coterie's DOM programming model is the Shared Object object replication package, which was designed to be integrated with Modula-3 and to address the needs of highly interactive, distributed applications.

To further ensure that simple tasks are easy, Coterie includes the most common building blocks needed for the sorts of applications we envision, the most important and interesting being a novel distributed 3D graphics library called Repo-3D. Finally, all of these facilities are made available in Repo, Coterie's high-level, interpreted, distributed language. Repo is the only interpreted language we know of that supports both client-server and replicated data uniformly across its entire type system. Using Repo, relatively sophisticated applications can be built and tested with a minimum of effort. The implementations of Repo and Repo-3D also serve to highlight the power and flexibility of the Shared Object package.

To ensure that "complex things are possible," Coterie is a general purpose programming environment whose data distribution facilities were designed with flexibility in mind. This is a key to its eventual success; because Coterie is being used to explore new computing paradigms, it should contain as few distribution or interaction *policies* as possible. Rather, the programmer is presented with languages and tools that are sufficiently powerful and flexible to experiment with whatever policies and approaches to data organization, control flow and user-interaction are appropriate. In effect, Coterie is a toolkit for building experimental AE systems, not an AE system itself.

This dissertation will discuss the design and implementation of the various components of Coterie, and how they fit together to satisfy our goal of an easy-to-use testbed for building distributed AEs. Examples of its use, and justification for the various choices, will be provided by presenting simple examples, and discussing complete applications, built using the toolkit.

1.2 Research Contributions

The research described in this dissertation involves the creation of a programming environment to support exploratory programming of distributed AEs, and in using this infrastructure to investigate some interesting problems. The contributions of this research are:

1. Shared Objects, a novel, tightly integrated replicated object package for a mainstream programming language (Modula-3) that presents the programmer with a powerful DOM programming model,
2. Repo, a distributed, interpreted language that presents a DOM to the programmer with both client-server and replicated data sharing semantics,
3. Repo-3D, a high-level, structured graphics library with directly distributable and extensible graphical objects,
4. Coterie, a testbed for fast prototyping of distributed AE applications that incorporates these components, and
5. A number of prototypes implemented in Coterie that explore different augmented environment application domains.

1.2.1 Shared Objects: A Distributed Shared Object Memory

The idea of distributed shared memory (DSM) was introduced by Li [Li, 1986, Li and Hudak, 1989]. His approach, and that of a number of others since then, was to implement shared memory at the operating system level, by leveraging the virtual memory architecture and integrating memory distribution with the paging system of the operating system. Unfortunately, this approach suffers from a number of problems, not the least of which is difficulty of implementation that arises because changes must be made at the operating system level.

An alternative approach, often referred to as Distributed Object Memory (DOM), is a distributed shared-memory abstraction that avoids the problems caused by the page-level granularity of DSM. In this approach, the illusion of one large shared memory is presented at the programming language level, by encapsulating the shared data in programming language objects and using the language constructs to ensure all access to that data is

through method calls on those objects. Access to objects is uniform: at the language level, local and remote objects are accessed in the same way. Distribution semantics (if an object is replicated, when to migrate single-copy objects, etc.) may be hidden, or language features may be used to control them.

While there have been a number of DOM systems, none are suitable for our work. All of the systems that support replicated data are part of non-mainstream programming languages (e.g., Argus [Liskov, 1988], Amber [Chase et al., 1989], Distributed Smalltalk [Decouchant, 1986], Emerald [Jul et al., 1988], Munin [Bennett et al., 1989], and Orca [Bal et al., 1992]). Those systems that are integrated with popular programming languages only support client-server data, with a single copy of each object and proxy-based remote access from other processes (e.g., RPC [Birrell and Nelson, 1984], CORBA [OMG, 1992], Network Objects [Birrell et al., 1993], Java RMI [Wollrath et al., 1996], and ILU [Janssen et al., 1998]). Unfortunately, client-server data is not sufficient for highly interactive applications. Replicated data is needed as well, since any data required to respond to a user's actions in real time must be local to the process handling that response. Since this replicated data may or may not need to remain synchronized over time, there are three basic data distribution semantics needed for our domain: client-server, unsynchronized replication, and synchronized replication.

Building a custom language, or working with an obscure one, is impractical, primarily because of the lack of existing software. One of the main reasons we chose Modula-3 [Harbison, 1992] for this work was the variety of software packages available for it. One particularly useful package provides an elegant solution for distributed shared objects with client-server distribution semantics (Network Objects [Birrell et al., 1993]). This package also ends up providing unsynchronized replicated objects though automatic marshalling of arbitrary data structures: any non-Network Object parameter or return value of a Network Object method call is automatically copied between processes. The Shared Object package is a complementary package that satisfies the need for tightly synchronized replicated data and is cleanly integrated with Modula-3 and the Network Objects package.

The Shared Object package design was inspired by the approach to object replication used by Bal and his colleagues [Bal and Tanenbaum, 1988, Levelt et al., 1992]. In their formulation, implemented in the Orca programming language [Bal et al., 1992, Bal et al., 1998], objects are replicated across machines as needed and the semantics of object replication are enforced by the language. Replication consistency is accomplished by write-update via *totally-ordered function shipping*, where the runtime environment ensures all methods that update an object are executed in the same order on all replicas of the object. As it turns out, this approach is extremely well suited to implementation as an add-on to a strongly-typed programming language such as Modula-3. Furthermore, the performance characteristics of this approach are appropriate for highly interactive graphical systems, where the objects tend to have a high read/write ratio and need low latency update distribution.

By encapsulating application state in the language objects and having the semantics enforced transparently, both Network and Shared Objects satisfy one of our primary goals by exhibiting a high degree of network data transparency. This is extremely important for exploratory programming, as changes in the distribution patterns of data (for example, when client-server data becomes replicated) will then have a minimal impact on the structure of the code. Because these objects are tightly integrated into the programming language, objects with different distribution semantics can be mixed in arbitrary ways with predictable, and sometime novel, results. These results have also been reported in [MacIntyre, 1995, MacIntyre and Feiner, 1996b].

1.2.2 Repo: A Distributed Interpreted Language

A common approach to rapid prototyping is to provide the programmer with an interpreted language in which they can build their applications: Coterie's interpreted language is called Repo. Since we want Coterie to support a common programming model throughout its various components, both the interpreted and compiled languages should support similar forms of object distribution.

Repo is based on a distributed language, called Obliq [Cardelli, 1995], that supports client-server distribution semantics of all data items (objects, arrays and variables)

via the Network Object package. Repo extends Obliq's type system uniformly so that all its data items can also be distributed using both unsynchronized replication semantics, via normal programming language objects, and synchronized replication semantics, via our Shared Object package. Repo also includes a number of libraries that are needed to support rapid prototyping in our domain, such as simple support for reflection, HTTP clients and servers, regular expressions and so on.

Unlike the Modula-3 DOM, in which only the programming language objects (and not other language data, such as arrays and sets) are distributable with all possible semantics, in Repo all data items (objects, arrays and variables) can take on any of the three possible distribution semantics. Since all of these data types can be mixed and matched in arbitrary ways, a wider range of interesting data structures can be developed in Repo than is possible in Modula-3. Because it allows distributed applications to be developed in a few lines of interpreted code, Repo turns out to be an excellent language for exploratory programming of distributed interactive applications. The implementation of replicated objects in Repo also serves to highlight the power of the programming model used by the Shared Object package.

1.2.3 Repo-3D: A Distributed 3D Graphics Library

Distributed graphics refers both to the architecture of a single graphical application whose components are distributed over multiple machines, and to systems for distributing the shared graphical state of multi-display/multi-person, distributed, interactive applications. Coterie is designed to support both of these architectures. The former is obviously supported by the combination of Repo and any non-distributed 3D graphics library; the latter is provided by Repo-3D.

While many excellent, high-level programming libraries are available for building stand-alone 3D applications (e.g., Inventor [Strauss and Carey, 1992], Performer [Rohlf and Helman, 1994], and Java 3D [Sowizral et al., 1998]), there are no similarly powerful and general libraries for building distributed 3D graphics applications. Programmers are typically forced to use a general purpose mechanism, such as Repo, for distributing application state, and then to manually synchronize that state at each site with the state of

a local graphics library. Keeping these “dual databases” synchronized is a complex, tedious, and error-prone endeavor. Repo-3D was designed to address this problem by allowing programmers to encode application state in its directly distributable and extensible 3D graphical objects.

Repo-3D is an object-oriented, high-level graphics package, derived from Obliq-3D [Najork and Brown, 1995]. Like Obliq-3D, Repo-3D’s graphics facilities are similar to those of other modern high-level graphics libraries. However, the objects used to create Repo-3D’s graphical scenes are directly distributable and extensible because they are built using Shared Objects. Repo-3D also takes advantage of the Shared Objects’ ability to allow programmers to locally modify the object’s replicated state. This addresses two concerns particular to distributed applications, interactivity and the frequent need for local variations to the global scene.

With most approaches to synchronized replicated data (including the Shared Objects), updates to distributed state will be slower than updates to local state, and the amount of data that can be distributed is limited by network bandwidth. Therefore, if interactive speed is not to be sacrificed, a programmer must be able to perform some operations locally. Additionally, there are times when a shared graphical scene may need to be modified locally for reasons other than efficiency. For example, a programmer may want to highlight the object under one user’s mouse pointer without affecting the scene graph viewed by other users. Repo-3D allows the properties of a graphical object to be modified locally, and parts of the scene graph to be locally added, removed, or replaced.

Repo-3D is the first implementation of a directly distributed and extensible 3D graphics library. In addition to providing a solution to some significant issues, we highlight a number of directions for future work both in distributed 3D graphics and in object-based distributed shared memory systems. As with Repo, the implementation of Repo-3D highlights the power of the Shared Object package. Repo-3D has also been reported to the research community in [MacIntyre and Feiner, 1998].

1.2.4 Coterie: Exploratory Programming of AE Systems

Coterie incorporates Repo and Repo-3D as its major building blocks, and is designed to support our experimentation with distributed AEs. Coterie provides a unique, general purpose environment for constructing experimental distributed virtual environment systems and applications in a heterogeneous computing environment[1].

In addition to Repo and Repo-3D, Coterie contains a set of building blocks particular to AE applications. These include support for typically useful virtual environment components, such as two, three and six degree-of-freedom (DOF) trackers and two-way differential constraints. More interestingly, Coterie includes a number of libraries (some of which have been developed based on our group's experiences implementing prototype applications) that demonstrate how a general purpose DSM can be used for building distributed virtual environment applications. This work has also been reported in [MacIntyre and Feiner, 1996b, MacIntyre, 1997].

1.2.5 Prototype Augmented Reality Applications

During the course of this research, the testbed has been used by members of Columbia's Computer Graphics and User Interfaces Lab to build a number of prototype applications that demonstrate the utility of this work and point the way towards future research.

These prototypes include both new application areas and enhanced versions of our group's previous prototype systems, and cover a wide range of application domains, including:

- extending our previous *architectural anatomy* project, in which we allowed a user to view the support structures (columns, beams, etc.) inside the walls of a building [Feiner et al., 1995], to include both construction of new structures and maintenance of existing structures [Webster et al., 1996a, Webster et al., 1996b],
- a maintenance and repair task for telephone crossboxes. A telephone crossbox is where a telephone company wire bundle is patched through to subscribers. In this prototype,

1. Coterie currently runs on Solaris, IRIX, Windows NT, Windows 95 and Linux.

we allow notes to be spatially attached to groups of terminals, so that repair personnel can reserve terminals, denote bad or suspicious terminals, and so on [MacIntyre and Feiner, 1996b].

- an augmented reality tour guide, where an outdoor site (such as the Columbia campus, in our case) is augmented with interesting information displayed on either the user's HMD or handheld tablet display [Feiner et al., 1997],
- exploring interface concepts for the National Tele-Immersion Initiative, such as how to integrate other user interface tools (e.g. the integration of the Brown Sketch system [Zeleznik et al., 1991], demonstrated during the presentation of [MacIntyre and Feiner, 1998]),
- exploring the management of information across multiple, heterogeneous display devices. EMMIE has been developed to explore this issue, focussing on techniques for dealing with privacy issues is multi-user environments [Butz et al., 1998].

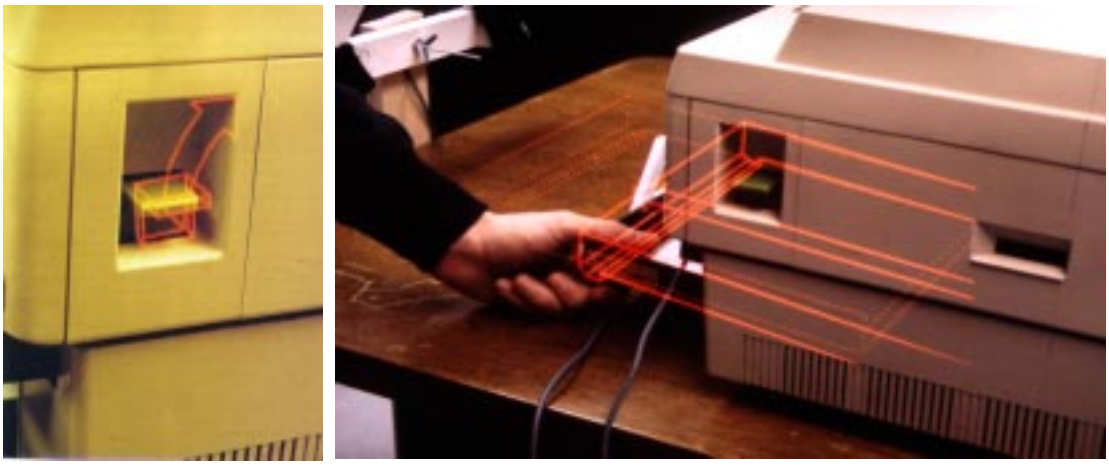
CHAPTER 2 An Overview of Coterie

In this chapter, we will give a high level overview of Coterie, the testbed we built to support our AE research, and which encompasses the ideas presented in this thesis. We will begin by detailing the high-level requirements that were identified for the testbed, based on our previous experience, both building distributed AR systems and using a number of VE systems. After presenting related work in distributed virtual environments and distributed groupware, we will discuss the design of Coterie in the context of our requirements, the related work and the practical implications of the environment in which it was built.

But first, a note about the name of the testbed. A *coterie* is a group of people who share a common interest. As will become apparent in this chapter, this concept fits well with our view of the kinds of distributed applications we will build, as a set of threads of control that are designed to work together to achieve a common goal. This is fairly important, as it prejudices our design decisions in favor of techniques that facilitate tightly-coupled distributed processes, in contrast to the decisions we would have made if we approached our applications differently. For example, if we built our systems as groups of agents, our design would favor techniques appropriate for loosely-coupled, self-contained threads. Coterie is also an acronym: the Columbia Object-oriented Testbed for Exploratory Research in Interactive Environments.

2.1 Previous Work: Augmented Reality

Our previous research in AR focused on stand-alone applications [Feiner et al., 1993a, Feiner et al., 1993b, Feiner et al., 1995]. The three prototypes we will discuss here are typical of what we and others envision to be some important potential uses of augmented reality. These prototypes are KARMA (maintenance and repair) [Feiner et al., 1993b], Windows on the World (integrating two- and three-dimensional information) [Feiner



(a) “Pull up the lever” (b) “Pull out the paper tray”

Figure 2-1: The KARMA prototype. Two images showing graphical instructions designed by KARMA to augment the user’s view of the printer. In both images, the system is conveying an action to be taken by the user, either lifting the printer lid lever or removing the paper tray. In both cases, the system highlights the object of the action and draws an animated arrow indicating the action to be taken. In (b), the system also draws a ghosted representation of the desired final location of the tray.

et al., 1993a] and Architectural Anatomy (examining hidden and alternate views of real world objects) [Feiner et al., 1995]. Each prototype was a stand-alone application assembled from an ad hoc combination of client-server software components.

2.1.1 KARMA

KARMA (Knowledge-based Augmented Reality Maintenance Assistant) was a prototype system that used a see-through head-mounted display to explain simple end-user maintenance for a laser printer. One of the key design goals of KARMA was to generate virtual information that complements the real world on which it is overlaid, taking advantage of what the user can already see. For example, one of the rules used by the system states that if a goal is to show the user where an object is located, the system must determine if the object is blocked by other objects. If it is blocked, it will be displayed so that it appears to be seen through the blocking objects; if it is already visible in the real world, it need not be drawn at all. In this prototype, we monitored the position and orientation of several key components of the printer by attaching 3D trackers to them and feeding this information into a modified version of the IBIS rule-based illustration generation system [Seligmann

(a) (b)

Figure 2-2: The Windows on the World prototype. The two views are shown through the user's head-worn display. The user is looking at another person in the lab who is tracked via a magnetic tracker worn around their neck. In (a), the person is working on a laptop, and in (b) both the user and the person in the image have moved to the left of the laptop. There are three windows visible in this scene: a note attached to the person, a small "load average" attached to the upper corner of the laptop display, and a *display fixed* control panel window for the hypermedia system (the window at the bottom). Notice that, as the person moves from working on the laptop to sitting to the left of it, the control panel window does not move, but the others do.

and Feiner, 1991] to interactively design the graphics and simple textual callouts that make up the virtual world. Figure 2-1 shows some images created with our prototype.

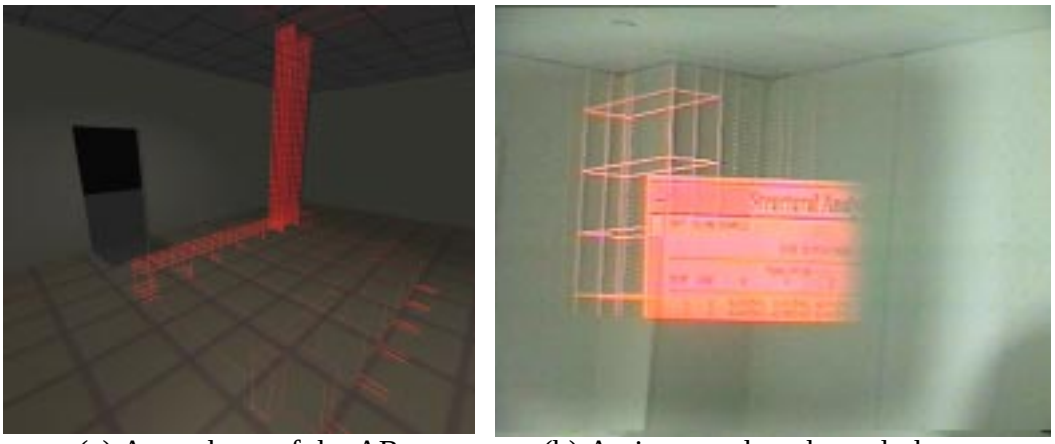
2.1.2 Windows on the World

Windows on the World [Feiner et al., 1993a] was our first attempt at integrating 2D text and graphics into a 3D virtual world. As with the KARMA system, the goal was to present virtual information that built on, and leveraged, the real world perceived by the user of the system. At the time, when people thought of using head-mounted displays and 3D interaction devices to present virtual worlds, it was usually in terms of totally synthetic environments populated solely by 3D objects. There are many situations, however, in which 2D

text and graphics of the sort supported by current window systems can be useful components of these environments, especially when this information is being merged with the real world. To explore this idea, we developed support for a full X11 window system server within our augmented reality testbed. The user head orientation was used to index into a large X bitmap, which effectively placed the user inside a display space mapped onto part of a surrounding virtual sphere. By also tracking the user's body, and interpreting head motion relative to it, a portable, see-through information surround was created that enveloped the user as they moved about. In this system, we supported three kinds of windows implemented on top of the X server: windows fixed to the head-mounted display, windows fixed to the information surround, and windows fixed to locations and objects in the 3D world. We also supported the ability to track arbitrary objects, allowing windows to move with them. To demonstrate the utility of this model, we developed a small hypermedia system that allowed links to be made between windows and windows to be attached to objects, as shown in Figure 2-2.

2.1.3 Architectural Anatomy

The Architectural Anatomy prototype was a collaboration between our group and the Building Technologies Group in Columbia's Graduate School of Architecture. In this project, we exposed a building's "architectural anatomy," allowing the user to see its otherwise hidden structural systems [Feiner et al., 1995].The prototype application overlaid a graphical representation of portions of a building's structural systems over a user's view of the room in which they were standing, as shown in Figure 2-3. The overlaid virtual world typically showed the outlines of the concrete joists, beams, and columns surrounding the room. In addition, because we built on the Windows on the World prototype's support for combining 2D X11 windows and 3D graphics in augmented reality, the system also allowed information about these support structures (such as the structural analysis of the column in Figure 2-3(b)) to be spatially positioned near the structure.



(a) A mock-up of the AR information display.

(b) An image taken through the see-through head-worn display.

Figure 2-3: The Architectural Anatomy prototype. Two images showing a column in the corner of our lab. Image (a) is a mock-up done by a design student (Ed Keller), showing a vision of what the system might display, while (b) is an actual image created by the system, including a 2D window containing a structural analysis of the column spatially attached to the column's center.

2.2 Motivation

Before starting work on this testbed, we had attempted to extend a number of our single-user, client-server prototypes (described in the previous section) in a variety of originally unforeseen ways: Architectural Anatomy was built on top of some components of the Windows on the World system, which was in turn built on top of some of KARMA's components. Unfortunately, that approach was becoming increasingly infeasible with each modification. These prototypes were built using a then common style of distributed programming, in which each logical component of the system is implemented as a separate process, and for any given application, the necessary components are linked together. Communication between the processes was done with a combination of custom RPC and message passing (both to replicate data that was needed in multiple processes, and to ensure that the data was distributed quickly without the need for polling of the data servers). Figure 2-4. shows the architecture of one of our systems [Feiner et al., 1993b][1]. Unfortunately, adding new processes to systems such as this often involved the need to

1. See the paper for a more complete description of each of the components.

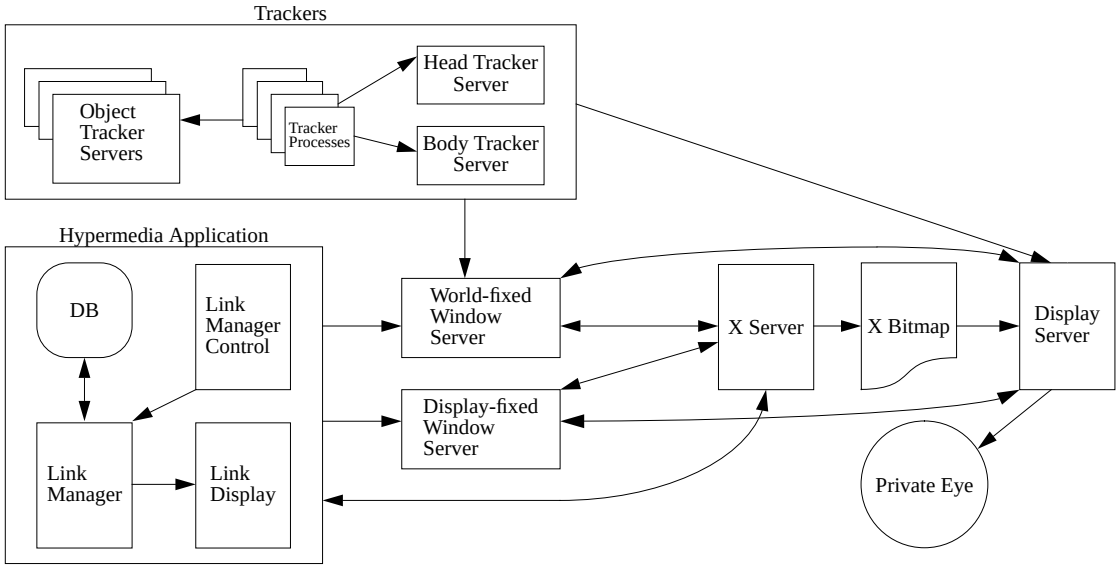


Figure 2-4: An example architecture diagram. The architecture of the Windows on the World system [Feiner et al., 1993b]. Each of the small labeled rectangles represents a process in the system (the larger rectangles represent logical groups of processes). Most of these processes are RPC and message stream servers. Many of them are also RPC and message stream clients. The arrows show data flow.

share previously unshared data, over time turning many of the clients into RPC servers and/or message stream sources. This resulted in an unmanageable welter of client-server relationships, with each of a dozen or more processes needing to create and maintain explicit connections to each other and to explicitly handle the inevitable crashes. This web of connections can be seen in Figure 2-4, where each rectangle is a process and the arrows show the information flow.

We spent a sufficiently large portion of our time reengineering client-server code that it became clear that (at least our implementation of) the client-server model was unsuitable, by itself, for exploratory programming of distributed virtual environment research prototypes. The heart of the problem was a lack of support for data sharing that was both efficient and easy for programmers to use in the face of frequent and unanticipated changes. Other problems we identified included the overhead of prototyping with compiled programming languages, the unsuitability of heavy-weight processes for implementing fine-grained parallelism and a lack of high level tools for building new applications. Our group was not alone in our frustration. For example, Pausch's group at the University of Virginia tackled the problem of rapid prototyping by creating the Alice

system [Pausch et al., 1995]. Alice, which is aimed at non-distributed, totally immersive VEs on the Windows 95 platform, includes fine-grained parallelism and an interpreted language. Alice is designed for use by world developers, and provides very high-level tools, whereas Coterie is aimed at system developers and provides more general, lower-level tools. Therefore, while the Alice designers do not concern themselves with distribution or with support for complex, multi-user applications, they do go farther than us in simplifying the development of single-user virtual environments: for example, they have a polished graphical interface for building worlds, whereas Coterie contains only APIs.

2.3 Requirements for the Testbed

The requirements we set forth for the testbed reflect our group's desire to experiment with distributed, multi-user AEs that combine a variety of paradigms, including opaque, see-through, desktop, and handheld displays. To effectively explore different design alternatives for this new paradigm, it is essential that programmers are able to create robust prototypes quickly and easily. Therefore, the infrastructure to support this exploration should satisfy the following requirements.

Data replication. Many of the objects in a VE or AE system must be replicated, rather than merely shared, because the programs using the data cannot afford to pay the price of remote access. A good example is the description of a graphical scene. The programs that update the displays must redraw their scenes as often as possible. Programs that do collision detection or other time sensitive computations must likewise access their databases on a continual basis and are often themselves distributed over multiple machines.

Uniform treatment of data. To build a distributed system, some data-sharing mechanism, such as remote procedure or method calling, is needed. Our experience has demonstrated that creating a distributed system that provides facilities for distributing only "virtual environment data," such as tracker readings or graphical objects, is far too restrictive. By treating some kinds of data differently than others, we occasionally found ourselves in the situation where one piece of data we needed, such as a tracker record, could

be easily distributed to a new component of the system, but another piece of data, such as the layout of a user's information space, could not.

Furthermore, the data types seen by programmers should have a high degree of *network transparency*: the programmer should be aware that a data value is not local to their machine only when absolutely necessary, and should be able to use remote and local data objects interchangeably whenever possible. Furthermore, the system should enable programmers to share resources and data objects easily and efficiently and have multiple threads of control in one or more processes concurrently access these shared resources. To provide this level of transparency requires the data sharing system to be tightly integrated with the programming language.

Responsive asynchronous data propagation. Remote procedure or method calls are unsatisfactory for propagating rapidly changing information because they are synchronous, and are therefore too slow, even when used with a small number of clients. The system should therefore provide a method of asynchronous data propagation, preferably one that would scale well as the number of distributed processes increased.

Asynchronous update notification. When many threads distributed over many processes share data, it is unacceptable for them to have to poll the data to check for changes. Instead, there must be some facility for interested threads to be notified of changes to relevant data items. For example, the thread that renders a graphical scene should be automatically notified of changes to the data structure representing the scene.

Embedded interpreted language. As has been demonstrated by a number of VE systems [Bricken and Coco, 1994, Pausch et al., 1995, Singh et al., 1995], if a system is to support rapid prototyping, the programmer should be provided with an embedded general-purpose interpreted language in which entire applications can be developed, without writing compiled code. While byte-compiled languages, such as Java [Arnold and Gosling, 1998], ameliorate some of the overhead of compilation, they do not support interactive modification of running programs in the same was that interpreted languages do.

Furthermore, since it will occasionally be necessary to rewrite some code in the compiled language (often for efficiency), the interpreted and compiled components of the system should be tightly integrated, should have similar programming models, and all data structures should be equally accessible from both. Our preference is to have both languages be strongly typed, either statically or dynamically, so the programming language can be leveraged as much as possible to assist programmers in creating reliable and robust programs.

Object-oriented and multithreaded environment. Conceptually, AE systems are composed of many independent objects that perform tasks such as monitoring trackers, rendering to displays, and controlling the entities that populate the environment. These map well to an object-oriented, multithreaded environment. However, using heavy-weight processes for all threads of control is unacceptable because of communication and context switching overhead. Creating processes that are inherently multithreaded without programming language or operating system thread support is error prone and requires considerable work to ensure all conceptual threads are serviced fairly. Furthermore, adding new conceptual threads in this fashion can be extremely difficult. Therefore, thread support should be integrated into the interpreted and compiled programming languages so threads may be used cleanly and uniformly across all operating systems and architectures.

High-level, platform-independent, extensible, 3D graphics package. It is essential that the environment support a wide variety of hardware and operating systems without the application programmer having to use a different graphics package on each. Furthermore, we want to be able to cleanly integrate new kinds of graphical objects, such as the workstation windows of [Feiner et al., 1993a].

Other desirable distributed system characteristics. In [Coulouris et al., 1994], the authors assert there are six key characteristics which determine the usefulness of a distributed system: resource sharing, scalability, openness, concurrency, fault tolerance and network data transparency. As discussed above, network data transparency is one of our primary requirements.

While the remaining features were deemed less important when trade-offs had to be made, they are obviously still desirable. Most importantly, the system should be designed to potentially scale well as more users, or processors per user, are added, and be open (if not to external programming languages, at least within the context of the environment) so that it can be extended in new and interesting directions. Basic fault tolerance is an absolute requirement: as the number of machines and processes per machine increases, so does the likelihood that one of them will crash, especially during development. At the very least, the system should allow programmers to construct applications that can recover from a single failure in a reasonably straightforward manner.

2.4 Related Research Areas

In this section, we shall look at approaches that have been taken to supporting distributed applications in two areas that are close to our application domain: virtual environments and groupware. Other areas of related work will be discussed in the chapters to which they are more directly relevant.

2.4.1 Virtual Environment Systems

A large number of VE toolkits have been created, of which only those that are intended to support distributed environments will be discussed: MR [Shaw and Green, 1993], DIVE [Carlsson and Hagsand, 1993], VEOS [Bricken and Coco, 1994], SIMNET [Calvin et al., 1993], NPSNet [Zyda et al., 1992], VERN [Blau et al., 1992], VR-DECK [Codella et al., 1993], WAVES [Kazman, 1993], RING [Funkhouser, 1995], BrickNet [Singh et al., 1995], dVS [Grimsdale, 1991] and Spline [Waters et al., 1997].

MR implements a simple shared virtual memory model. Raw memory locations can be marked as shared and local changes explicitly flushed to the other copies, which must then explicitly receive the changes. MR has no facilities for handling heterogeneous architectures and provides a single, fully replicated VE, in which each process has a complete copy of the same world. DIVE is built on top of the ISIS [Birman, 1993] fault-tolerant distributed system, and is similar to MR in supporting only fully replicated VEs. VEOS is an extensible environment for prototyping distributed VE applications. MR,

VEOS, and DIVE all use point-to-point communication, with all processes directly connected to all others. This prevents these systems from scaling beyond a relatively small number of distributed processes.

SIMNET is perhaps the best known large-scale distributed VE system. It uses a well-defined communication protocol (DIS) that is also used by NPSNet and VERN. SIMNET was designed to support a single, large-scale, shared, military VE. Broadcasting is used to send messages between nodes. While this cuts down on network traffic, all processes must handle all messages, preventing SIMNET from scaling beyond a few hundred users. NPSNet has recently been extended to accommodate a significantly larger number of simultaneous users (thousands instead of hundreds) by spatially partitioning its world to reduce message traffic [Macedonia et al., 1995]. Unfortunately, the SIMNET protocol is not general enough for our use, nor is a flexible SIMNET client program available.

VR-DECK allows multiple users to share a single simulation on a set of homogeneous workstations, but cannot be easily extended to support heterogeneous workstations. Message traffic is reduced by sending events only to machines known to be interested in them, but all machines potentially talk to each other, reducing scalability.

WAVES uses message managers to mediate communication between processes. Each message manager controls a group of clients. All messages are distributed by the message managers to interested clients. WAVES supports the ability to filter messages to a given client, reducing the type and frequency of updates sent. However, it supports only coarse parallelism, with each process performing one well-defined function. Its single shared world comprises a set of objects that encapsulate the behavior and state of the entities in the world. Each object is owned and updated by only one client, but can move freely between clients.

RING and BrickNet both use a communication mechanism similar to that of WAVES, with centralized servers each controlling a set of clients, and communication routed through the servers. All message traffic goes through the servers, with no provision made for direct client-client propagation for time-critical data. RING is geared toward realistic simulations and uses physical visibility to limit message traffic. Its VE is a set of

shared entities, each with a geometric description and a behavior. Each entity is owned by one client, and only that client may update it. RING can support a large number of simultaneous users. BrickNet is geared toward creating multi-user distributed VEs in which each client has its own world composed of a combination of local and shared objects. Like WAVES, its objects have behaviors as well as state and can move between clients.

dVS is a commercial distributed VE system for single-user applications. Its components and message formats are fixed and not extensible, making it unsuitable for non-exclusive VEs.

Spline is a system that adopted object sharing and data flow features similar to ours, and is aimed at efficient creation of immersive VEs. It achieves scalability by spatially partitioning its distributed database to reduce message traffic, starting with a scheme similar to [Macedonia et al., 1995] and extending it by partitioning the object space based on this spatial partition. While Spline has powerful data replication facilities, they do not provide the level of network data transparency we desire, primarily because Spline takes a distributed database, rather that distributed language, approach to object distribution.

None of these systems came sufficiently close to supporting enough of the features we needed to justify attempting to extend them to support the rest. With the exception of DIVE and Spline, none provide true preemptive threads, but use only heavyweight UNIX processes. With the exception of BrickNet and Spline, none support more than a single shared VE. In addition, these systems are geared toward VEs in which each user has only a single (stereo) display, and interacts with an entirely virtual world composed of 3D objects. In contrast, consider the hybrid window manager [Feiner and Shamash, 1991], a simple example of the kind of application we would like to support. It combines a flat-panel display with a see-through head-worn display to create a workspace with one display's image embedded in the other's. This would be difficult to implement with any of the VE systems mentioned here.

2.4.2 Distributed Groupware

A number of groupware systems have been built using shared object techniques. Colab [Stefik et al., 1987] uses a fully replicated database in which changes are broadcast to all sites without synchronization. Colab relies on social and application solutions to avoid, or recover from, inconsistencies. For example, if inconsistencies arise when multiple people are working on the same area of a document, they will quickly become obvious because of the nature of these applications. The users can then decide how best to deal with them.

GroupKit [Roseman and Greenberg, 1996] applications run the same program at all sites and communicate by using *multicast remote procedure calls* to execute procedures at all sites. Data is shared via shared data directories called *environments*. While supporting notification of the addition, deletion or modification of items in an environment, there is no support for concurrency control. As with Colab, social solutions are relied upon to solve this problem.

Object World [Tou et al., 1994] implements shared objects in LISP, and defines shared operations by allowing programmers to define broadcast methods. These methods are executed at any site that has a copy of *any* of the object parameters, with all additional parameters automatically copied to that site. Object World does not provide any consistency guarantees, but accomplishes consistency detection by requiring that all broadcast methods operate on the same version of their object parameters at all sites. Correction of inconsistencies is performed at an application level, possibly with the assistance of the user.

DistView [Prakash and Shim, 1994] allows window and application objects to be replicated. When an object is replicated, it is wrapped in a proxy that implements the replication semantics, such as sending method invocation messages to remote copies for execution. There is no distinction between read and write methods, and consistency is guaranteed by requiring locks to be acquired for all object accesses. While DistView has a fairly intelligent scheme to minimize the cost of acquiring global locks, the system would not scale well and would not perform well in the face of continuous access from multiple sites.

The techniques for object sharing implemented in the newer groupware toolkits share some of our goals, particularly automatic replication of data to ease construction of distributed applications (e.g., Prospero [Dourish, 1996]). However, none have integrated the distribution of data into the object model of their respective programming languages as tightly as we desire. Furthermore, many of them do not provide sufficiently strong consistency guarantees. In groupware applications, inconsistencies tend to arise from multiple users attempting to perform conflicting actions: the results are usually obvious to the users and can be corrected using social protocols. This is not an acceptable solution for VE applications. Finally, none of these object systems provide any support for asynchronous update notification, nor are they designed to support the kind of large scale distribution we have in mind.

2.5 Implementation Overview

Coterie was written in the Modula-3 programming language [Harbison, 1992]. The decision to use Modula-3 was based on the language itself and the availability of a set of packages that provided a solid foundation on which to base our research.

Modula-3 is a descendent of Pascal that corrects many of its deficiencies. In particular, Modula-3 retains strong type safety, while adding facilities for exception handling, concurrency, object-oriented programming, and automatic garbage collection[2]. One of its most important features for our work is that it gives us uniform access to these facilities across all architectures. The availability of three packages strongly influenced our decision to use Modula-3:

- **Network Objects.** The Network Object package [Birrell et al., 1993] supports a client-server model of distributed data sharing through remote method calls that are virtually transparent to the programmer. These include distributed garbage collection, exception

2. The Modula-3 compiler was developed at DEC's (now Compaq's) Systems Research Center. We used a commercially supported version of the SRC compiler, developed by Critical Mass, Inc. and distributed as part of the Reactor programming environment. The compiler, and thus our system, runs on all the operating systems we use: Solaris, IRIX, Linux, Windows NT and Windows 95.

propagation back to the calling site, and automatic marshalling and unmarshalling of method arguments and return values of virtually any data type. We also enhanced this package to provide automatic data conversion between heterogeneous machines, as this facility (although advertised) did not exist. (Our enhanced package is distributed with both the commercial and free versions of the Modula-3 compiler.)

- **Obliq.** Obliq [Cardelli, 1995] is a lexically-scoped untyped language for distributed object-oriented computation that is tightly integrated with Modula-3. Like Modula-3, it supports multiple threads of control within a single process. Obliq's distributed computation mechanism is based on Network Objects, allowing transparent support for multiple processes on heterogeneous machines. Objects are local to a site, while computation can roam over the network.
- **Obliq-3D.** Obliq-3D [Najork and Brown, 1995] is a high-level 3D animation system that consists of two parts: a Modula-3 library that provides a basic set of 3D graphical objects and animation primitives, and those same primitives embedded in Obliq. Obliq-3D programs can be written in Modula-3, Obliq, or any combination of the two because all data structures are simultaneously available from both languages. Obliq-3D's structure and interface also make it relatively easy to extend.

Together, these packages provided a good starting point for our work. Modula-3 and Obliq gave us a compiled and interpreted language with closely matched programming models supporting object-oriented, multi-threaded programming. Network Objects (and therefore Obliq) also provided a clean basis for reliable distributed programming via a simple client-server DOM based on transparent remote method calls. Finally, Obliq-3D gave us the high-level, platform independent 3D graphics library we required.

The development of Coterie took place on two fronts. Initially, a (primarily) single-process testbed was created that enabled our group to begin building prototypes, some examples of which will be discussed in Section 2.6. Simultaneously, the facilities for building distributed applications were designed and implemented so that the techniques and packages built using the initial version would extend naturally into the distributed domain. This latter work is the topic of the bulk of this dissertation, and can be divided into three major parts. First, the need for transparent data replication was satisfied through

the creation of a replicated object system, called the Shared Object package, that also provides asynchronous update propagation and notification of changes to the replicated objects. When combined with the Network Object package, the resulting DOM satisfies the majority of our data sharing needs. The Shared Object package is the topic of Chapter 3. The second component of this work, discussed in Chapter 4, involved using the Shared Object package to extend Obliq to support replicated data, resulting in a new interpreted language called Repo (for Replicated Obliq).

The final component of this research was to create a distributed graphics library called Repo-3D, a redesign of Obliq-3D that fits cleanly within the DOM programming model presented to the programmer in Modula-3 and Repo. Since both the development time and code structure of the prototypes being built are dominated by the manipulation of the graphical scenes, Repo-3D simplifies development by making all graphics objects directly distributable and extensible and adding facilities to support the peculiarities of building distributed graphical applications. Repo-3D is discussed in Chapter 5.

2.5.1 Virtual Environments: Tracker Support

One major component of Coterie that is needed to facilitate the creation of (even non-distributed) virtual environment applications, which we will draw on in subsequent chapters as a source of examples, is support for various tracking systems. The tracker package was initially built in Modula-3 and exposed into Obliq, with the intent of eventually building Repo wrappers around the tracker objects to support easy distribution of tracker data. The distribution of tracker reports serves as the basis for a simple example of the use of Shared Objects in Chapter 3. We return to this example in Chapter 4, both to show how simple replicated objects are created in Repo, and as a vehicle for illustrating more complex behaviors. Finally, in Chapter 5 we show how the location of a tracker can be embedded directly in a graphical scene and therefore distributed transparently. The Repo help files for all of the modules in the Tracker package are contained in Appendix B.

To support the goal of providing modular, reusable components, a generic tracker object and a hierarchy of tracker report objects were developed. The aim is to encapsulate

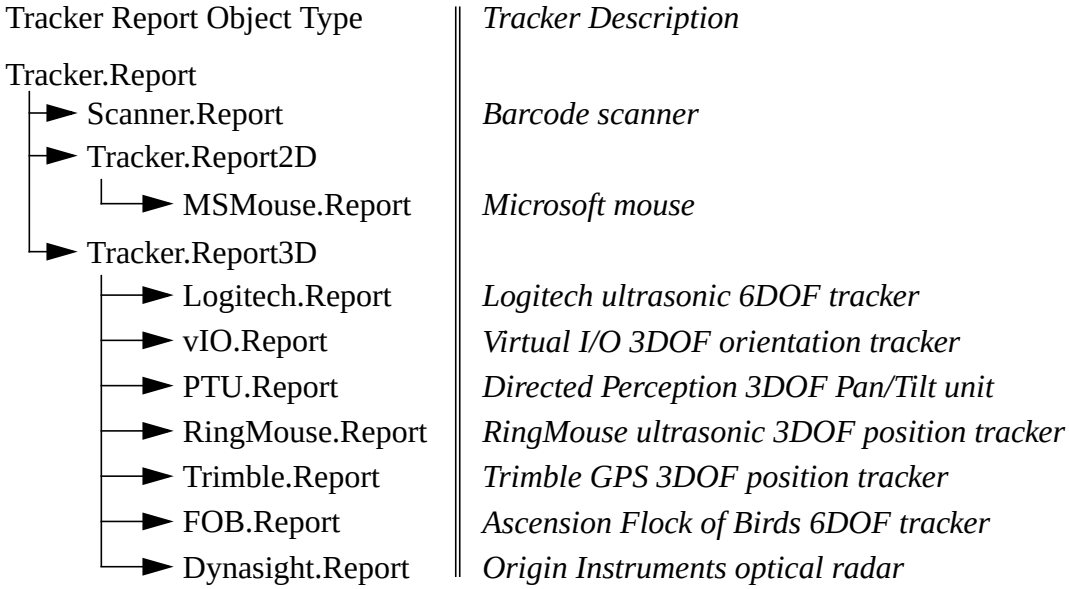


Figure 2-5: The generic Tracker Report Object hierarchy. The basic tracker `Report` object is returned by the generic `Tracker.T` object and many clients. Those that specifically need two or three dimensional reports would use the next level in the hierarchy, and those that need to know details of the specific devices would use the reports specific to the devices. More objects will be added to the hierarchy as more devices are supported.

support for all our tracking systems into one common object hierarchy, as done by other systems such as the UNC Tracker Library [Holloway, 1991]. The generic `Tracker.T` object, from which any object representing a tracking device inherits, supports a set of common methods: `read()`, `reset()` and `close()`.

The `read()` method of any `Tracker.T` object returns a `Tracker.Report` object, which is the root of a hierarchy of objects representing progressively more specialized kinds of tracker reports. The current `Tracker.Report` hierarchy is shown in Figure 2-5, along with a description of the tracking devices that return them. Clients are written to use the most general kind of report they can. The dynamic type system of Modula-3, Obliq and Repo allows clients to be written that accept the generic `Tracker.Report` and handle specific parts of the hierarchy differently. The most common case is that a client accepts a reasonably specific object, such as the `Tracker.Report3D`, and does something useful with it. For example, a client that associates a three dimensional position with an object *Obj* would accept

`Tracker.Report3D` objects and position the object *Obj* in the virtual world relative to the position of the 3D tracker. Such a client would be able to use all devices that report 3D positions.

2.6 Initial Prototypes

As mentioned in Section 2.5, our group continued its AR research while Coterie was being developed, building a number of (mostly) non-distributed new prototype systems. We will close this chapter by discussing four of them: a reimplementation of the Architectural Anatomy system, a maintenance application for telephone company crossboxes, a space frame construction assistant for the Augmented Reality for Construction (ARC) project, and the Touring Machine, a mobile tour guide.

These systems demonstrate the utility of the initial version of Coterie for developing simple AE prototypes. The first three prototypes were each implemented in a few hundred lines of code, illustrating the ease with which ideas can be explored in Coterie. The last prototyping is the most complex prototype we implemented with the non-distributed version of Coterie, and illustrates how the system supports complex systems to be built as well.

2.6.1 Architectural Anatomy

The first non-trivial program built with Coterie was a reimplementation of the Architectural Anatomy system, discussed in Section 2.1. This version of the system is different from the previous version (Figure 2-3) in two ways: the implementation is much simpler, and the graphical representation of the architectural structures is more complex. The simplicity of the implementation is due to the power of both the infrastructure and the hardware the system runs on, allowing the entire prototype to be implemented in a few hundred lines of code and run on a single machine. Since Coterie supports full color 3D graphics, we provide more graphical cues to help the user spatially locate the architectural structures behind the walls of the room, as shown in Figure 2-6. The images of the columns show a representation of the column as well as the reinforcement bars, or “rebar”,

Figure 2-6: The new Architectural Anatomy prototype. This image shows two columns in a corner of our lab.

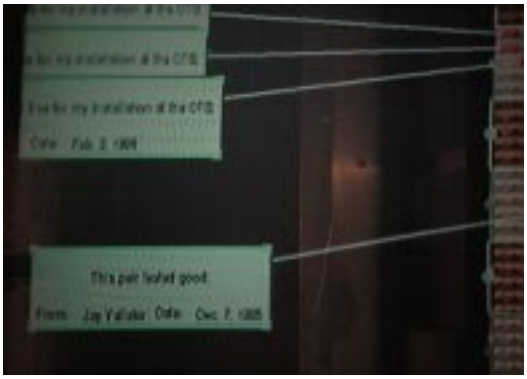
inside the column. The transparent box representing the column extends from the ceiling to the floor, helping the user place the column visually behind the room walls. The walls, floor and ceiling of the room are also shown transparently, to further help orient the user.

2.6.2 Telephone Crossbox Maintenance

The one feature that was missing in our reimplementation of the Architectural Anatomy system was the integration of 2D windows, that allowed us to include external 2D information such as the structural analysis of the column shown in Figure 2-3. We added this facility into Coterie when we began to explore how AR could be used for maintenance of telephone company crossboxes, in cooperation with Nynex Science and Technology. Drawing on discussions with Nynex workers, we found that one useful application of AR is to allow 2D information to be attached to groups of terminals inside the crossbox, as shown in Figure 2-7. The notes could be used by the workers to make virtual “post-it” notes, or by the system to communicate information to the workers. For example, the system could ensure that all terminals are periodically tested by attaching notes to those that have not been tested recently, prompting the workers to test them at their convenience.

(a) The bottom of a phone company “crossbox” that connects customer phone lines to company wiring. The terminals in half of the box are connected to phone company wires, and those in the other half are connected to subscriber lines. No wires are connected in this picture.

(b) The top of the crossbox with a graphical overlay designed to be presented to the field service person on the head-worn display.

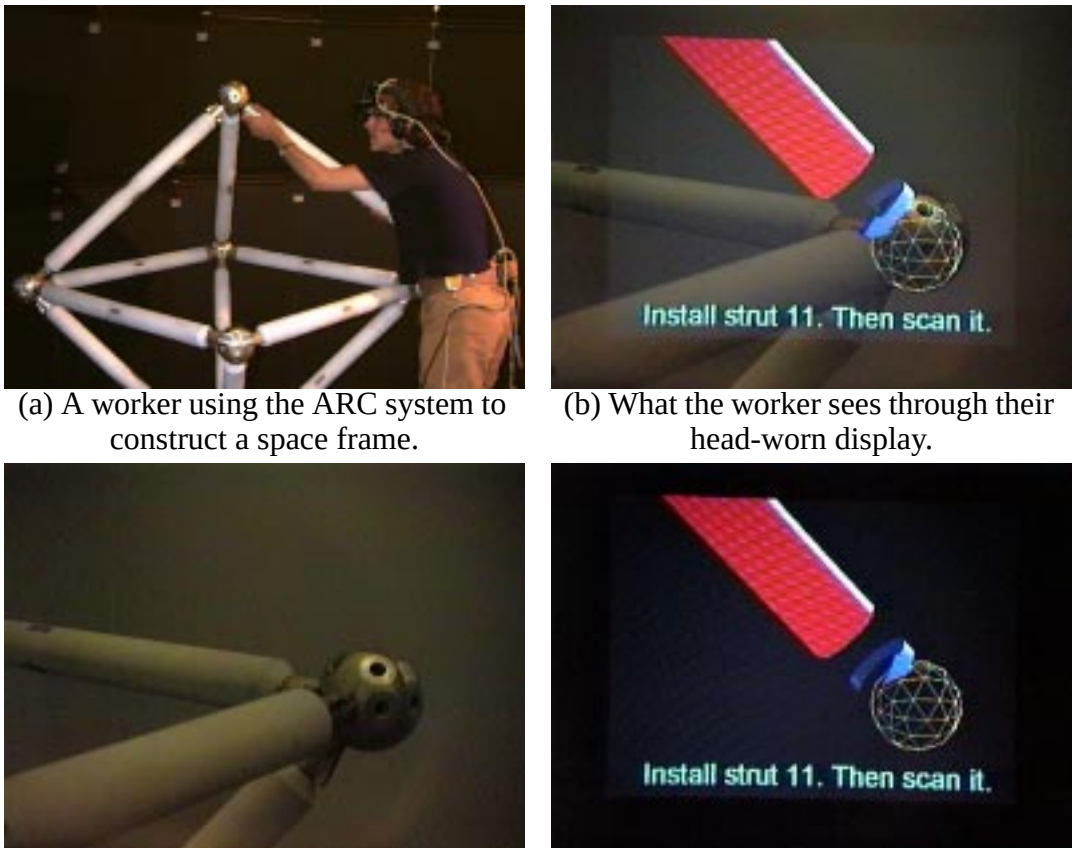


(c) The view when the user looks down and to the left from (b).

Figure 2-7: A prototype AR application for crossbox maintenance. The overlay highlights major blocks of the crossbox and a number of user-defined groups of connection posts. It also contains 2D information windows connected to the post groups by stretchable leader lines that allow selected windows to be pulled into and out of view.

2.6.3 Spaceframe Construction

As part of the Augmented Reality for Construction (ARC) project, we built an AR system to assist with the construction of space frame buildings. Our system prompted the worker by displaying the next part to be installed in the correct location on the partially completed space frame, as shown in Figure 2-8. Like the Architectural Anatomy system, this proto-



(a) A worker using the ARC system to construct a space frame.

(b) What the worker sees through their head-worn display.

(c) The real world

(d) The virtual world

Figure 2-8: A prototype AR application for space frame construction. This system was built with the non-distributed version of Coterie, and is designed to lead a worker through a construction sequence to ensure the correct pieces are installed in the correct locations in the proper sequence. (a) shows the system in use. (b) shows what a worker would see when using the system. The elements of the real and virtual world that are combined to form the image in (b) are shown (c) and (d), respectively.

type ran on a single computer and took a few hundred lines of code to implement. The initial version of this prototype was built by an undergraduate student with no prior experience using Coterie, over the course of a few months.

2.6.4 Automated Tour Guide

The Touring Machine prototype [Feiner et al., 1997] was designed to assist the user in exploring various kinds of information as they move freely about a relatively large environment (in this case, the Columbia University campus). By displaying information in context using a combination of displays (in this case, a see-through head-worn display and

(a) The user wears a backpack and headworn display, and holds a handheld display and its stylus.

(b) Labels increase in brightness as they near the center of the display.

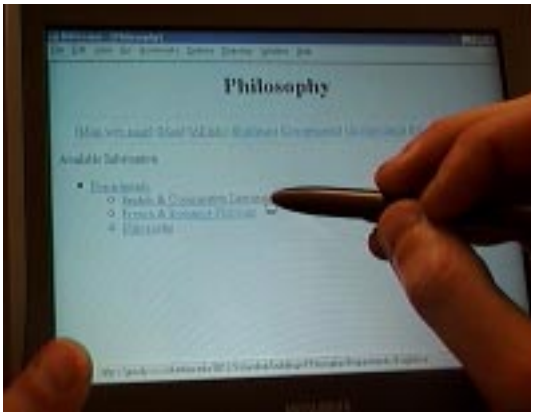
(c) The Philosophy Building with the "Departments" menu item highlighted.

Figure 2-9: A prototype campus information system. The physical prototype is shown in (a), while (b) and (c) show images of campus buildings with overlaid names, shot through the see-through headworn display. The handheld display has a trackpad on the back that can be used to select from the context-sensitive menu presented at the top of the head-worn display.

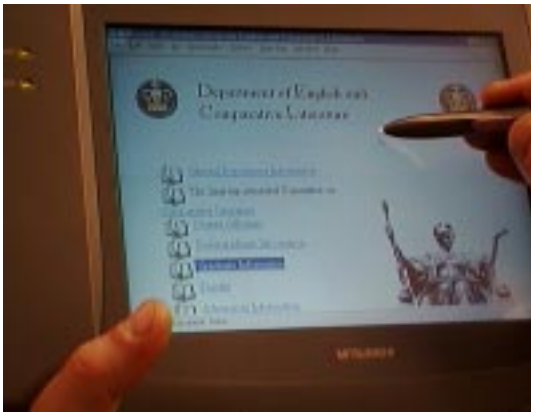
a hand-held tablet computer, as shown in Figure 2-9(a)) and allowing the user to select from a small set of currently relevant information cued to their current location and interests, the system allows the user to explore a complex space in a relatively straightforward manner. The system was designed to operate in an environment of low precision position tracking (provided by differential GPS) and take advantage of the different characteristics of the two display devices. For example, we label buildings, not specific building features, overcoming registration problems that would otherwise occur, as shown in Figure 2-9(b). Furthermore, when a menu item is selected, the detailed information about that item is presented on the (easier to read) handheld. To inform the user when this display transition is occurring, the selected menu item label is animated to fly toward the handheld, as shown

(a) After the “Departments” menu item in Figure 2-9(c) is selected, the department list for the Philosophy Building is added to the world, arrayed about the building.

(b) The image in (a) also shows the beginning of the label animation sequence, a fraction of a second after selection. Here is the animation, approximately half a second later.



(c) Selecting the “Departments” menu item causes an automatically-generated URL to be sent to the web browser on the handheld computer, containing the department list for the building.



(d) Actual home page for the English and Comparative Literature department, as selected from either the generated browser page or the department list on the handheld web browser.

Figure 2-10: Additional images of the Touring machine. (a) through (c) illustrate the results of selecting the “Departments” item from the context-sensitive menu for the Philosophy building: the department list is added to the world near the building in (a), an animated label flies off the bottom of the display, starting in (a) and continuing in (b), and the list of departments is presented on the handheld in an automatically generated web page. If the user selects one of the departments, they are taken to its web page in (d).

in Figure 2-10(a) and (b). The handheld displays information via a web browser, as shown in Figure 2-10(c) and (d).

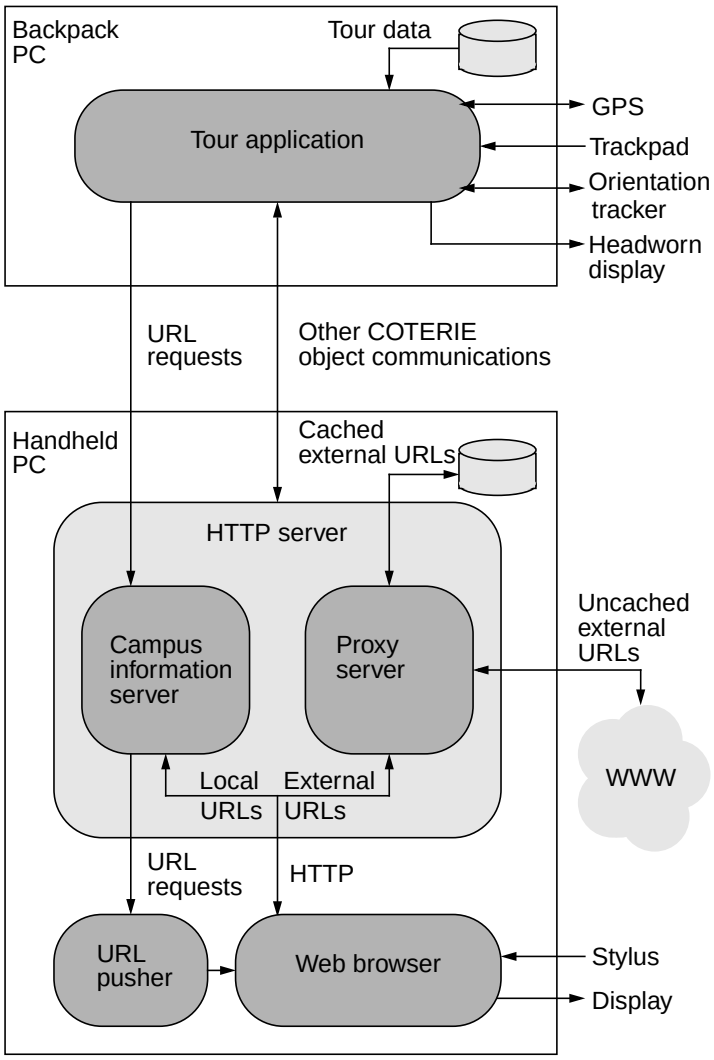


Figure 2-11: Software design of the prototype campus information system. There are two instances of Coterie, one running on each of the two machines (labelled "Tour Application" and "HTTP server"). The URL pusher and Web browser are external programs. The two web servers in the HTTP server application are embedded within an Obliq program that is tightly integrated with the tour application via client-server object sharing. The campus information server is responsible for the dynamic generation of HTML pages, and the caching proxy server exists to mitigate the slowness of the radio network link.

This prototype is the first distributed application built with Coterie. Since it was built prior to Repo, it does not take advantage of any data replication. It comprises two applications, one running on each of the two machines, implemented in approximately 3600 lines of commented Obliq code. Figure 2-11 shows the overall software structure. The *tour application* running on the backpack PC is responsible for generating the graph-

ics and presenting it on the headworn display. The application running on the handheld PC is a custom *HTTP server* in charge of generating web pages on the fly and also accessing and caching external web pages by means of a proxy component. By running our own HTTP server on the handheld display, we can react to user input from the web browser and head-worn display simultaneously in a straightforward manner. For example, when a URL is selected on the handheld display, the HTTP server can call a network object method that selects corresponding graphical items on the headworn display. Thus data selection works in both directions: from the backpack PC to the handheld PC (by launching relevant URLs from the headworn display's menus) and vice versa (selecting buildings, departments, etc. on the headworn display from a link on the handheld's browser).

Even though there is a relatively small amount of data sharing going on between these two programs, we did run into the need to replicate data between the two: both applications needed a copy of the Tour Data database (shown near the Tour application in Figure 2-11). Fortunately, in this simple prototype, the data does not change while the program is running, so we arranged to copy the database from the Tour application to the HTTP server application when the system is started. While this is a far from ideal solution, it serves to illustrate the need for flexible and general-purpose data replication.

CHAPTER 3 Shared Objects

As was discussed in the previous chapters, one of the primary motivations for designing Coterie was to create an environment that presents an easy to understand model of distributed data sharing to the programmer, cleanly integrated into a mainstream programming language. An obvious way to make distributed programming easy to understand is to present a model of data sharing that is familiar to the audience and compatible with the style of programing in which they typically engage. Distributed Shared Memory (DSM), and Distributed Object Memory (DOM) in particular, satisfy these criteria. However, providing the programmer with a conceptually easy-to-understand model is only part of the solution; we must also present it to them in a way they can use, by integrating it with a familiar programming language. In our case, the language is Modula-3, but the techniques discussed here are equally applicable to other languages, such as Java [Arnold and Gosling, 1998] or C++ [Ellis and Stroustrup, 1992].

The utility of presenting a DOM programming model via a tightly integrated distributed programming package has proved to be very useful, and is not new: many packages exist that present the programmer with client-server data distribution in this way (e.g., Network Objects [Birrell et al., 1993] and Java RMI [Wollrath et al., 1996]). Others exist that present a less integrated model with the same basic purpose (e.g, RPC [Birrell and Nelson, 1984], CORBA [OMG, 1992], ILU [Janssen et al., 1998]). However, none of these packages provide facilities for object replication, especially replication that is fast and geared toward the needs of highly interactive, graphical applications such as ours. It is this problem that the Shared Object package addresses.

We will begin this chapter by presenting the concepts of DSM and DOM in Section 3.1, and discuss other systems that have been built using this model in Section 3.2. In Section 3.3 we will discuss the design of the Shared Object package, beginning with a

summary of the benefits provided by tightly integrating an object replication package with a strongly typed language such as Modula-3. In this section we also present the details of the totally-ordered, write-update replication model we use, and discuss the facilities provided to allow programmers to be notified of changes to a local replica of a Shared Object. This latter facility is important for our applications, as it enables an event-driven control flow that obviates the need to poll objects, looking for changes.

In Section 3.4 we will discuss the implementation of the Shared Object package. In this section, we present a detailed example of the package in use, and use this example to discuss the various implementation choices we made. Finally, in Sections 3.5 and 3.6 we discuss the performance and usability of the system, and our observations and experiences using Shared Objects to implement interactive application prototypes.

3.1 Distributed Shared Memory

DSM allows a network of computers to be programmed much like a multiprocessor, since the programmer is presented with the familiar paradigm of a common shared memory. DSM mechanisms use message-passing protocols between machines to implement some model of shared memory access that is used by the programmer.

The idea of DSM was introduced by Li in his doctoral work [Li, 1986], in a system called Ivy [Li and Hudak, 1989]. DSM presents the programmer with the illusion that the memory of all machines in the distributed system belongs to one large shared address space. The approach used in Ivy, and that of a number of other subsequent systems, was to implement shared memory at the operating system level, by leveraging the virtual memory architecture and integrating memory distribution with the paging system. Unfortunately, this approach suffers from a number of performance problems, as well as difficulty of implementation. The two fundamental problems with page-based DSM are:

- Memory coherence, and therefore distribution, is at the granularity of a page, which may not match the granularity of application data. It falls to the applications to solve the problems of granularity and placement of shared data in order to avoid false sharing of data and the associated unnecessary network messages this implies.

- Application access patterns do not guide the DSM coherence mechanism that is replicating and invalidating pages, so optimizations such as prefetching and relaxed memory consistency are less effective.

The lack of a programming model that allows application semantics to influence data sharing is the Achilles Heel of page-based DSM.

An alternative approach, often referred to as Distributed Object Memory (DOM), presents the illusion of one large shared memory at the programming language level, by encapsulating the shared data in programming language objects and using the language constructs to ensure all access to that data is through method calls on those objects. The distributed address space is partitioned implicitly by the application programmer, with an object being the smallest unit of sharing. Typically, references to objects exist only in processes that are interested in the object. Unlike page-based DSM, semantics may vary on a per object basis: some objects may be replicated, while others could exist at one site with proxies at other sites that access the single copy via remote method invocation. Access to objects is uniform: at the language level, local, remote and replicated objects are accessed in the same way. Distribution semantics (whether an object is replicated, when to migrate single-copy objects, etc.) may be hidden, or language features may be used to control them. By implementing the distribution mechanisms at the application level, the problems caused by page-level granularity in DSM are avoided. The advantages of this model over techniques that expose the shared memory at a lower layer are discussed further in [Levelt et al., 1992].

While there have been a number of DOM systems built over the years, none are suitable for our work. On one hand, most of the systems are part of non-mainstream programming languages (e.g., Argus [Liskov, 1988], Amber [Chase et al., 1989], Emerald [Jul et al., 1988], Munin [Bennett et al., 1989], and Orca [Bal et al., 1992]). On the other hand, those systems that have been designed to work with mainstream languages have all supported client-server data, where a single copy of each object exists at some site and all other sites have a proxy that performs (more or less) transparent remote access to that single copy (e.g., RPC [Birrell and Nelson, 1984], Distributed Smalltalk [Decouchant, 1986], Network Objects [Birrell et al., 1993], Java RMI [Wollrath et al., 1996], CORBA

[OMG, 1992], ILU [Janssen et al., 1998]). Unfortunately, client-server data is not sufficient for highly interactive application domains such as ours. Replicated data is needed as well, since any data required to respond to users actions in real time must be local to the site processing that response. However, since not all replicated data needs to remain synchronized over time (for example, it may be immutable), and the synchronization protocols add overhead to data access, we need to support both synchronized and unsynchronized replicated data. Therefore, there are three basic data distribution semantics needed for our domain: client-server, unsynchronized replication, and synchronized replication.

Building a custom language, or working with an obscure one, is not feasible, primarily because of the lack of existing software that would be available: one of the main reasons we chose Modula-3 [Harbison, 1992] for this work was the variety of useful software packages readily available for it. In particular, the Network Objects package provides an elegant solution for distributed objects with client-server semantics [Birrell et al., 1993]. This package also provides unsynchronized replicated objects though automatic marshalling of arbitrary data structures: any non-Network Object parameter or return value of a Network Object method call is automatically copied between processes, creating a new copy (replica) of the data that has no further relationship to the original copy. The goal of our work on the Shared Object package was to create a complementary package that satisfies the need for tightly synchronized replicated data, that is cleanly integrated with Modula-3 and the Network Objects package, and is designed with the needs of our application domain in mind.

The Shared Object package described here meets this goal. It is tightly integrated with the language, using a compile-time code generator that takes annotated Modula-3 source code as input and generates the necessary code to provide strictly consistent replicated objects. The design was inspired by an approach to object replication used by Bal and his colleagues [Bal and Tanenbaum, 1988, Levelt et al., 1992]. In their formulation, implemented in the Orca programming language [Bal et al., 1992], objects are replicated across machines as needed and the semantics of object replication are enforced by the language. Replication consistency is accomplished by write-update via *totally-ordered func-*

tion shipping, where the runtime ensures all methods that update objects are executed in the same order on all replicas of the objects. As it turns out, this approach is extremely well suited to implementation as an add-on to a programming language (see Section 3.3). Furthermore, the performance characteristics of this approach are appropriate for highly interactive graphical systems, where the objects tend to have a high read/write ratio, need local reads to be fast, and demand low latency update distribution.

By encapsulating application state in the language objects and having the semantics enforced transparently, both Network and Shared Objects satisfy one of our primary goals by exhibiting a high degree of network data transparency. This is extremely important for exploratory programming, as changes in the distribution patterns of data (for example, when client-server data needs to becomes replicated data) should then have a minimal impact on the structure of the code. Because they are tightly integrated into the programming language, objects with different distribution semantics can be mixed in arbitrary ways with predictable, and sometime novel, results. Furthermore, the Shared Object package was designed with an eye towards accommodating additional semantics in the future, as they are identified. This is extremely important, as initial experience has born out our suspicion that additional semantics would be needed after we gained experience using the system. Examples of additional semantics will be discussed in Chapters 4 and 5. These results have been reported in [MacIntyre, 1995, MacIntyre and Feiner, 1996b].

3.2 Related Work

The majority of work on distributed, object-based programming systems has focused on client-server semantics [Decouchant, 1986, Jul et al., 1988, Bal et al., 1992, Bennett et al., 1989, Birrell and Nelson, 1984, Birrell et al., 1993, Wollrath et al., 1996, OMG, 1992, Janssen et al., 1998]. One of the reasons that client-server data packages are common is that the distribution semantics are straightforward and relatively simple to implement; each object exists at one site, and all other copies access that object remotely. While the model can be complicated by support for additional features, such as caching and object migration, the fundamental concept remains simple. Packages supporting data replication, on the other hand, have a variety of possible semantics and design trade-offs,

	Coterie					
	Shared Objects	Network Objects	Penumbra	Orca	Dist. Smalltalk	Emerald
client-server		✔	✔	✔	✔	✔
caching proxies	n/a		✔			
synchronized replication	✔			✔		
unsynchronized replication	✔	✔				✔
user-defined consistency			✔		n/a	n/a
object migration	n/a		✔	✔	✔	✔
update	✔	n/a	✔	✔	n/a	n/a
invalidation		n/a	✔		n/a	n/a
mainstream language	✔	✔	✔		✔	

Table 3-1: A comparison of distributed object-based programming systems. Coterie includes both Shared and Network Objects, allowing it to support both replicated and client-server data sharing. Caching proxies, such as offered by Penumbra [Kristensen and Low, 1995], can be thought of as a kind of replication, but do not support automatic data propagation, and the first read of an object (after any change) requires network access. As can be seen, only Orca and Coterie support both client-server and replicated data, but Coterie is integrated into a mainstream programming language. The table also shows that the areas of improvement for Coterie are in the area of user-defined consistency and object migration.

and even the simplest useful designs are significantly more difficult to implement than a client-server package. Table 3-1 shows a comparison of some representative distributed programming environments.

Emerald [Jul et al., 1988] is a typical example of a language designed from the start with distribution in mind. Like most of the early distributed languages (e.g., Argus [Liskov, 1988]), it supports client server distribution of objects. Unlike earlier languages, it presents the programmer with a uniform programming model for both local and remote object manipulation. It also supports object migration.

RPC [Birrell and Nelson, 1984], CORBA [OMG, 1992], ILU [Janssen et al., 1998], Distributed Smalltalk [Decouchant, 1986], Network Objects [Birrell et al., 1993] and RMI [Wollrath et al., 1996] are all client-server distributed programming toolkits that are either designed to work with, or are enhanced versions of, existing languages. RPC, Corba and ILU are designed to be language independent, whereas Network Objects, Distributed Smalltalk and RMI are designed to work with a specific language (Modula-3, Smalltalk and Java, respectively). Because they are tightly integrated with a

single programming language, these toolkits typically provide the features of that language on a distributed scale, such as distributed garbage collection, exception propagation between sites, support for marshalling of complex arguments, and so on. Distributed Smalltalk also provides facilities for object migration.

While there have been a number of languages created that support replicated data (e.g., Orca [Bal et al., 1992], Distributed Oz [Haridi et al., 1997], Mentat [Grimshaw, 1993]), most DSM systems are implemented as libraries that can be linked with programs written in an existing sequential language. A large number of parallel and distributed languages exist that extend sequential object-oriented languages such as C++ (many are discussed in [Wilson and Lu, 1996]). Mentat is a good example. Unlike our approach, and that of Orca, Mentat does not aim at tight integration with the object model of the C++. Instead, it lets programmers express what should be executed in parallel and uses a macro data-flow model to allow methods to execute in parallel and block access to variables that have not yet had values assigned to them. Like Mentat, Distributed Oz (which extends the Oz language) uses a programming model that is not tightly integrated into the object model of the language [Haridi et al., 1997]. It uses single-assignment logic variables and abstractions such as ports and cells to express distribution. In designing the distributed extensions to Oz, Haridi et al. were explicit in their choice not to integrate the distributed semantics transparently into Oz because they do not feel that it is possible to provide tight integration and efficient, fault-tolerant, scalable distribution mechanisms.

The most similar attempt to ours at creating a language preprocessor (or extension) to tightly integrate replicated data into a mainstream language is the Penumbra toolkit for C++ [Kristensen and Low, 1995]. Penumbra is based on the notion of Problem-oriented Object Memory (POOM), an object-oriented extension to the notion of problem-oriented shared memory [Cheriton, 1986], where application semantics are taken into account to relax consistency and improve distribution efficiency. Unlike Coterie, Penumbra does not provide symmetric replicated objects, but instead retains the notion of a single master object and supports caching the object data in the distributed proxies. Techniques are provided to manipulate those caches, to create methods that operate on the cache locally to minimize network message traffic, and for the master object to retrieve information from

the caches when needed. Their approach has the drawback that significant work is required by the programmer to implement subtypes of their base class `Distributable`, and a greater burden is placed on the shoulders of the programmers to ensure consistency. Furthermore, the objects are not truly distributed, and maintaining strictly consistent caches is less efficient than in Coterie. While their system is more flexible, the performance gains are aimed at objects whose read/write ratio is extremely low (i.e., primarily write operations) and is most dramatic on those objects. As they point out, the approach taken by languages such as Orca is well suited to objects whose read/write ratio is high. Penumbra is also not particularly well integrated into C++: only subtypes of `Distributable` can be distributed as parameters or return values to method calls, and specialized object-factories and process representations must be used by the programmer to build distributed programs.

The approach to replication most closely related to ours is Orca, a distributed language developed by Bal and others [Bal and Tanenbaum, 1988]. As discussed earlier, it is their approach to object replication that we used as a model for the Shared Object package. In Orca, like other DOM systems, shared state is encapsulated in objects and that state can only be accessed through object methods. An important characteristic of Orca objects is that all method accesses are atomic, creating a distributed programming model much like monitors in local shared memory. Unlike most DSM and DOM implementations, Orca uses a write-update protocol based on function shipping and totally ordered group communication: methods that update an object are applied to all replicas in the same order. Methods that do not change an object are applied only at the site that executed them.

An update protocol was chosen for implementing write operations over the more common invalidation protocol for a number of reasons. First, invalidation protocols are commonly used in page-based DSM schemes because each logical update to a data structure usually modifies multiple memory locations, and each modification is considered a separate update. Thus, using an update scheme would be wasteful, because a single logical update would result in significant unneeded network traffic as intermediate updates were distributed. Updates in Orca, however, are method calls and are therefore much more complex. Broadcasting these updates makes sense, and turns out to be relatively efficient,

especially if network broadcasting is used. Function shipping (where the arguments to a method are distributed and the method executed at all sites) was chosen over data shipping (where the method is applied once and the object state distributed) to avoid transferring the entire object state after each update, because, for the applications Bal is interested in, objects are often much larger than the arguments to the methods. To update replicated objects in a coherent way, each operation is sent using totally ordered group communication, so all updates are executed in the same order at all machines.

The Orca system supports both replicated and single-copy client-server objects. An interesting feature of Orca is that it replicates only objects that are expected to have a high read/write ratio, thus reducing the overhead of updates. The runtime can make this determination based on programmer specification, static compiler analysis, dynamic evaluation of the runtime behavior of the program, or a combination of the latter two. When the read/write ratio of the object is low, replication is inefficient, so the object in question is stored on a single machine, with other machines accessing the object via remote method calls. We will discuss the Orca system further throughout this chapter.

While the main difference between Orca and the Shared Object package is that the Shared Object package is designed as an extension to an existing language, they also differ in a number of other ways. For example, in both systems, total ordering is enforced through the use of sequencers (see Section 3.3.2), but in Orca a running application has one sequencer, while the Shared Object package allows any number to be used and distributes data hierarchically through them. This is the only general-purpose replication system of this kind that we know of that supports multiple sequencers, although a number of distributed virtual environment systems have architectures of this sort (see Section 3.3.2). Another important difference is that the Shared Object package supports notification of changes to replicated objects, which Orca does not. Finally, the Shared Object package is much more flexible in practice than a language such as Orca. This is partially by design, and partially an outgrowth of its implementation as a language add-on, which allows programmers, when necessary, to circumvent the restriction that data must be accessed through methods. This latter feature ends up being quite important when large systems are

being built, as will be seen in the discussion of the implementation of Repo-3D in Section 5.5.

3.3 Shared Object Package Design

The primary goal of the design of Shared Object package is to provide a structured, strongly-typed method to replicate state that is tightly integrated with Modula-3 and the Network Object package. In this section we will discuss our reasons for choosing a distribution mechanism that is tightly integrated with our programming language and describe the model of totally ordered, write-update objects in greater detail. We will also describe the Callback Objects that provide us with support for notification of changes to Shared Objects and discuss the communication architecture of the system.

3.3.1 Goal: Tight Integration

The decision to tie distribution to programming language objects was driven by the benefits this approach provides the programmer, none of which are specific to VE programming:

- Debugging distributed applications is non-trivial. Taking advantage of the type system of a strongly typed language helps ensure that distributed data is used correctly.
- Tight integration raises the level of network transparency and simplifies learning the system because there is only one programming model to master.
- Data is passed between processes as arguments to, or return values from, method calls. Therefore, the programmer has direct control over what data is distributed where.
- Arbitrary data structures can be passed between processes, and synchronized replicated objects can be embedded within other data structures. Therefore, the programmer does not have to be directly aware of all replicated objects for them to be distributed. Rather, logical data structures are distributed as a unit, and changes to the internal organization of those data structures does not necessarily require changing how they are used.
- Only objects that are referenced in a process have their state replicated in (and updates sent to) that process. When all references to an object are dropped at a given site, it is

garbage collected and the replica removed. The runtime notices and updates are no longer delivered to that process.

In addition to these anticipated benefits, the design was also influenced by the Modula-3 environment for which it was being developed. In particular, the Network Objects client-server data-sharing package already existed and provided a clean solution to client-server data distribution. Since we wanted to use this package, it is important that the Shared Object package provide a similar experience to the programmer. This was realized by following the design of the Network Object package in three ways:

- Shared Objects are tightly integrated with the full range of Modula-3 language features, including supporting distributed garbage collection and handling exceptions in a clean and obvious way.
- Shared Objects behave as much like normal programming language objects as possible.
- Shared Objects are defined by inheriting from a certain, distinguished object type provided by the package and following a few programming conventions. To enforce the distribution semantics, a compile-time code generator checks these conventions and creates related objects.

3.3.2 Model: Totally Ordered, Write-Update Objects

Recall that the important attributes of the Shared Object approach to data replication are the use of an object as the most basic unit of distribution, and of a write-update protocol based on function shipping and totally ordered group communication to enforce consistency.

By designating an object as the smallest granularity of distribution, each object is either fully replicated at a site or not; there is no support for the notion of partial replication of an object. Adopting this restriction greatly simplifies the implementation of object replication, while not sacrificing much power in practice. When used in conjunction with the Network Object package, partial replication can be mimicked using a combination of the two types of objects. Indeed, since most of the application development with our system is done in Repo (see Chapter 4), where all objects (aside from basic types such as inte-

gers or booleans) can be Network or Shared Objects, creating hybrid objects that combine both is straightforward.

A write-update consistency protocol satisfies the requirements set out in Chapter 2:

- A write-invalidation protocol violates the requirement of fast read access to replicated data because the first read that takes place after the local replica is invalidated requires a network access to fetch the updated object.
- This extra network access slows down the propagation of updates through the system, which violates the need for low latency data propagation.
- While an invalidation protocol allows the programmer to be notified that a local replica has changed, it is not possible to provide any additional semantic information about what has changed, as this information is contained in the update. Without this semantic information, the programmer would have to examine the object, requiring it to be fetched from the network immediately, degrading the system to an extremely inefficient simulation of a write-update protocol.

The semantics of this model are embodied in the following two principles, that also describe the Orca system:

1. All operations on an instance of an object are *atomic* and *serializable*. All operations are performed in the same order on all copies of the object. If two methods are invoked simultaneously, the order of invocation is non-deterministic.
2. Property 1 applies to operations on single objects. Making sequences of operations atomic is up to the programmer.

The property of serializability allows an important simplification to be made to Property 1. By distinguishing between methods that update an object (update methods), and those that do not (read methods), the read methods can be executed locally and the object replicas remain valid. Since reads are always executed locally, the model guarantees fast read-access to shared data. Furthermore, this model fits the design considerations discussed in Section 3.3.1 quite well, and is quite easy for programmers to understand and use: all methods are executed locally in the order they are called by the programmer, and all methods that update an object are executed everywhere in the same order. For example,

if two update methods are invoked simultaneously at different sites, the order of execution of the two updates is non-deterministic, but is guaranteed to be the same at both sites.

The implicit atomicity of method calls further enhances the understandability and predictability of program behavior because no explicit locks are required. The importance of this observation should not be underestimated: without atomic method invocation, the likelihood of programmers implementing unreliable objects with hidden race conditions is quite high.

It should also be noted that while the model says much about the order of execution of methods, it says relatively little about the actions performed by those methods (except to distinguish between methods that update the object state or not). This is possible because of the use of function shipping, as opposed to data shipping. This choice is important both for the reasons given in Section 3.2, and for the following reasons:

- Since the state of an object is not shipped around after each update, the state does not need to be examined after the updates, which would be a tricky and difficult endeavor in a language postprocessor. The alternative, examining the data itself after each update, can be extremely time consuming, especially if an object contains complex data structures. Without analyzing the side- effects of the methods, the entire state would have to be shipped each time, which would be inefficient.
- Data shipping implies that the only important side effect of method application is the changes to the internal state of an object. However, when a programmer is watching an object for changes, knowing what methods are executed, and what the arguments to those methods where, often provides sufficient information to react to that update without analyzing the contents of the object.
- Data shipping implies that all object state is global. As will be seen in the examples in Sections 4.6.4 and 4.6.5, and in the implementation of Repo (Section 4.7) and Repo-3D (Section 5.5), there is often a need for non-replicated, “local” state to be maintained within the objects.
- Data shipping implies there is internal state that can be shipped. With function shipping, stateless replicated objects can be used to mimic structured message ports, where

calling update methods corresponds to message distribution. Since message passing is a commonly used approach in distributed virtual environments, being able to mimic it is important, since flexibility, not dogmatic adherence to any one programming model, is our goal. Examples of using a replicated object as a structured message port can be found in Sections 4.6.1 and 4.6.3.

Given these factors, it is no surprise that we know of no existing distributed VE systems that uses an invalidation-based replication scheme. A hybrid update/invalidation scheme would give some additional benefit, such as allowing programmers to choose to trade off update latency for reduced network utilization, but would have significantly increased the implementation complexity of the system.

To ensure all update methods are applied to all replicas of an object in the same order, the messages containing the data for each update method (the method identifier and the arguments) are delivered to each replica using totally-ordered group communication. While there are many approaches to group communications [Birman, 1993, Coulouris et al., 1994], the Shared Object package adopts the same approach used in Orca. A total order on updates to an object is enforced by having all updates to that object send an event to a distinguished process called the *sequencer.* The sequencer for an object assigns a sequence number to each update event it receives in the order it receives them, and forwards the update messages to all processes containing a replica of that Shared Object. Updates are applied only after they are received from the sequencer. Any process may serve as a sequencer for one or more objects, in addition to its other tasks.

Currently, each process is associated with a particular sequencer and all objects created in that process have their updates sequenced by that sequencer. We call the set of processes associated with the same sequencer a *cluster.* The sequencer sends direct updates only to processes in its cluster. If a process in another cluster has a copy of this Shared Object, the update is sent to its sequencer, which will then forward the update to the processes in its cluster that have a copy of the object. Thus, all update messages between clusters are sent through the sequencers, which communicate directly with each other.

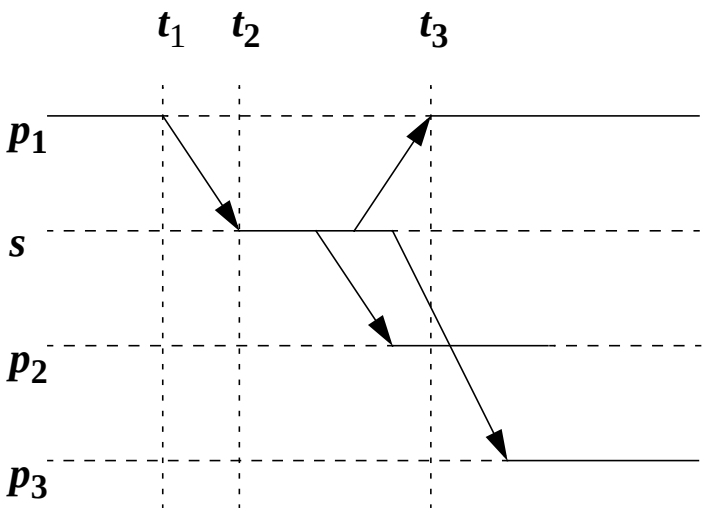


Figure 3-1: Control and data flow for a Shared Object update. Process p_1 calls an update method at time t_1. It blocks after a message is sent to the sequencer s. The message arrives at time t_2 and is sent to all interested clients (p_1, p_2 and p_3). p_1 receives the message back at time t_3, is unblocked, executes the original method call and continues.

Consider what happens when a thread initiates an update by calling an update method. The thread is blocked until the update event is received back from the sequencer, at which time it applies the update to the object and returns. Figure 3-1 shows the control and data flow of a typical update method call. A similar connection topology is used by many VR systems (e.g., WAVES, BrickNet, and Ring) to reduce network traffic, but none have isolated it in this fashion. For example, in BrickNet and Ring, the low-level functionality of the sequencer is combined with other high-level functions such object lookup and management. In WAVES, these functions are assigned to separate processes, but these processes exist in a one-to-one relationship with each other.

Overlapping high- and low-level functionality like this is undesirable. For example, all these systems have a one-to-one relationship between the number of sequencers and application level object management servers. In contrast, in Coterie application level object management can be partitioned based on conceptual process groupings, while sequencer duties can be partitioned based on physical network characteristics. These two partitions are not always identical, as illustrated in Figure 3-2.

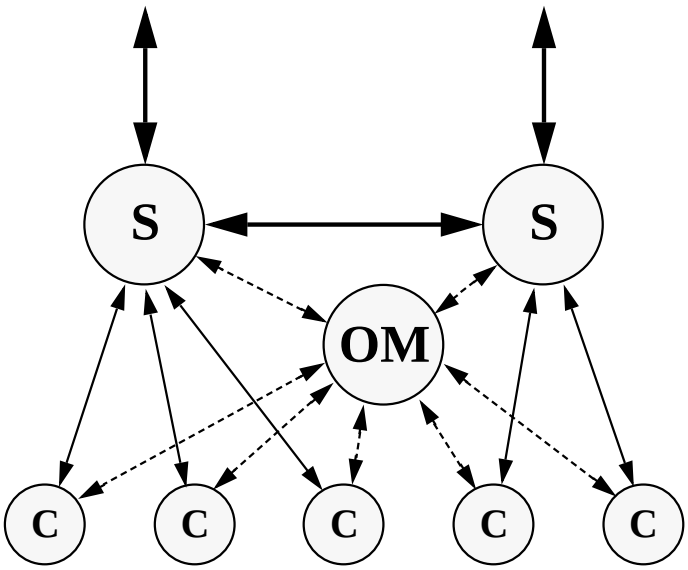


Figure 3-2: The relationship between clients, sequencers and object managers. The partitioning of clients (C) between sequencers (S) can be based on efficiency considerations, whereas the partitioning of clients between object managers (OM) can be based on the semantic grouping of processes, which are generally not the same.

3.3.3 Event Driven Control Flow: Callback Objects

In order to satisfy the need for asynchronous notification of changes to replicated objects, we introduce the notion of Callback Objects into the replication model. Callback Objects allow the programmer to receive notification of changes to a Shared Object in an structured fashion. For any Shared Object, the compile-time code generator creates an associated Callback Object that has a set of methods corresponding to the update methods of the Shared Object.

Callback Objects are used as follows. An instance *CO* of a Callback Object is associated with an instance *SO* of a Shared Object by passing *SO* to the constructor of *CO*. When an update method of *SO* is invoked, the corresponding method of *CO* is called. Since the internal representation of a Shared Object is hidden, the details of the change are indicated to *CO* by passing the arguments of the original method call on *SO* to the corresponding method of *CO*. To receive notification of an update, a programmer creates a new object and overrides the methods of *CO,* corresponding to the changes to *SO* for which notification is desired, with methods that react appropriately to those changes. Callback

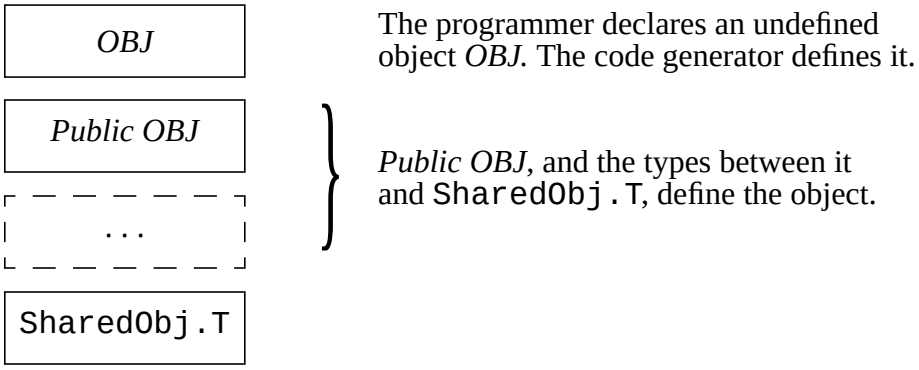


Figure 3-3: Object hierarchy for a Shared Object. The programmer implements the objects *Public OBJ* and all the supertypes up to `SharedObj.T`. The Shared Object code generator generates the Modula-3 code to implement *OBJ*, which overrides all the methods of *Public OBJ* to implement the Shared Object semantics.

Objects remove the need for object polling and enable a "data-driven" flow of control. A concrete example, showing how tracker date can be distributed and changes reacted to in a timely fashion, will be given in Section 3.4.1. Manny of the other examples of the usefulness of Callback Objects will be given in later chapters.

The Callback Objects contain two methods for each update method in the corresponding Shared Object. These methods can be overridden to receive notification before ("`pre_`" methods) or after ("`post_`" methods) an update to the Shared Object. An additional pair of "catch-all" notification methods can be overridden to receive notification, before or after an update, of any changes not handled by the more specific notification methods.

3.4 Implementation

There are two components to the implementation of the Shared Object package: the compile-time code generator, and the Shared Object Runtime. Before describing these, we will give an overview of the package from the programmer's viewpoint.

From the view of a Modula-3 programmer, a Shared Object type *OBJ* is created, as shown in Figure 3-3. The programmer defines *OBJ* as a subtype of some other Modula-3 object (*Public OBJ*) with no publicly accessible data fields. *Public OBJ* inherits (possibly

via other objects) from a distinguished object, `SharedObj.T`.[1] The programmer annotates the source code to tell the system which of the methods of *OBJ* should be treated as update methods. The Modula-3 source code implementing the Shared Object semantics is automatically generated and consists of the definition of *OBJ* that overrides the methods of *Public OBJ* to implement the shared object semantics, and a Callback Object for *OBJ* (see Section 3.3.3). The generated code also contains a set of proxy objects for type *OBJ* to facilitate embedding the objects in an interpreted language such as Repo.

Access to an object is controlled by a single writer, multiple reader paradigm. Update methods are executed on an object within an exclusive lock and non-update methods are executed inside a shared lock. This allows multiple non-update methods to execute in parallel, while ensuring that update methods have exclusive access to the object while they are updating its internal state. The locking scheme we implemented ensures that readers cannot hold off writers indefinitely.

3.4.1 Object Definition and Runtime Code Generation

When defining a Modula-3 Shared Object, there are some restrictions that the programmer must obey, which are quite similar to those imposed by the Network Object package:

- The programmer must leave a “hole” at the end of the object hierarchy, by declaring the last object to be a subtype[2] of some other object, and not defining this final type anywhere in the code. It is this type that is defined by the Shared Object code generator.
- The remaining objects in the hierarchy must not have any visible data elements declared in this interface, which is assumed to be the primary interface for the object. This is enforced because, according to the model, all access to the data fields must take place through the object methods.

---

1. In Modula-3, when a type in an interface is used, it must be qualified by the interface name. Thus, the `T` type in Figure 3-4 is used elsewhere as `TrackerPosition.T`.
2. Modula-3 supports the notion of defining an object as a subtype of another object without providing the details of this new subtype. Such subtypes are referred to as opaque subtypes and written using the notation “T<:S”, which is read as “T is a subtype of S.” These partial declarations can be revealed elsewhere in the code using the REVEAL statement.

- The object hierarchy must be rooted at the `SharedObj.T` type.
- All methods must be defined as raising the `SharedObj.Error` exception. If there is a problem with object communication, or if the replica has been rendered invalid, that exception will be raised.
- Subtypes of a Shared Object are not considered valid Shared Objects. If a programmer wishes to create a Shared Object that is a subtype of another Shared Object, the new object should inherit from the non-Shared Object (*Public Obj* above, and `Tracker-Position.S` in the example below) and have the code generator generate a new Shared Object for this type.

3.4.1.1 Example Object Definition

To make these restrictions clearer, let us return to Coterie's Tracker package, discussed in Section 2.5.1. The package defines two kinds of objects, subtypes of `Tracker.T` and `Tracker.Report`. The former are interfaces to the various tracking devices, and the latter are the reports periodically generated by these device interfaces. None of these objects, however, are Shared Objects. The device interfaces only exist in the process that communicates with the actual hardware device, and the reports are considered to be immutable data elements, similar to integers, characters or text strings.

To support distribution of the tracker reports throughout the system, the tracker package contains a simple Shared Object that implements a replicated container for tracker reports. The definition of the object, called a `TrackerPosition` for historical reasons, is shown in Figure 3-4.

The state of the object being defined contains a single tracker report. To distribute tracker reports, an instance of the `TrackerPosition.T` object is passed to all interested processes. When the thread that reads the tracker device obtains a new tracker report, it calls the `set()` method of this instance to update the replicated object state with the new report value. The implementation of this object, and a private interface that exposes the internal details of the implementation[3], is shown in Figure 3-5.

```
INTERFACE TrackerPosition;
IMPORT Tracker, SharedObj, Thread;

CONST Brand = “TrackerPosition”;

TYPE
   Data = Tracker.Report;

   T <: S;
   S <: Public;
   Public = SharedObj.T OBJECT
      METHODS
         init (): T RAISES {SharedObj.Error};
         set (READONLY val: Data) RAISES {SharedObj.Error};
         get (): Data RAISES {SharedObj.Error};
         <* SHARED UPDATE METHODS T.set, T.init *>
      END;

END TrackerPosition.
```

Figure 3-4: The Modula-3 interface definition for `TrackerPosition`. The public portion of the tracker object is defined to have three methods (`init`, `set` and `get`), of which one (`set`) is an update method on the object (`init` is also technically an update method, but it is generally only called during object creation). The file `TrackerPosition.i3` would contain this definition.

In this example, as with all our code, we will follow the Modula-3 convention that the main type in an interface (the one a programmer would use) is named `T` (thus requiring it be referred to as `TrackerPosition.T`, in this case). Before looking at the definition of the `TrackerPosition` object, consider a typical Modula-3 package containing a type `T` that the designer wishes to have both public and private parts. To partition the object, the designer declares that `T` is a subtype of a second type, `Public`, via the declaration:

---

3. Having the details of the implementation exposed in an interface is currently required because the code generator needs this information and it is not possible to extract it from a Modula-3 implementation (.m3) file. This is not a serious problem, because Modula-3 supports the notion of private interfaces that cannot be accessed outside a package.

```
INTERFACE TrackerPositionF;

FROM TrackerPosition IMPORT T, S, Public, Brand, Data;

REVEAL
   S = Public BRANDED Brand OBJECT
         data: Data := NIL;
       OVERRIDES
         init := Init;
         set := Set;
         get := Get;
       END;

PROCEDURE Init (self: S): T;
PROCEDURE Set (self: S; READONLY val: Data);
PROCEDURE Get (self: S): Data;

END TrackerPositionF.
```

(a) A private interface, `TrackerPositionF.i3`, containing the internal definition of the `TrackerPosition` object.

```
MODULE TrackerPosition EXPORTS TrackerPosition, TrackerPositionF,
                               TrackerPositionProxy;

PROCEDURE Init (self: S) : T =
  BEGIN
    IF self.data = NIL THEN
      self.data := NEW(Data);
    END;

    RETURN self;
  END Init;

PROCEDURE Set (self: S; READONLY val: Data) =
  BEGIN
    self.data := val;
  END Set;

PROCEDURE Get (self: S): Data =
  BEGIN
    RETURN self.data;
  END Get;

BEGIN
END TrackerPosition.
```

(b) The implementation of the `TrackerPosition` module, `TrackerPosition.m3`, containing the definition of the object methods.

Figure 3-5: The Modula-3 implementation for `TrackerPosition`. These two files implement the private parts of the `TrackerPosition` Shared Object; (a) a private interface and (b) the implementation.

```
TYPE
   T <: Public;
   Public = OBJECT ... END;
```

`Public` is fully defined in the interface and would contain the public parts of the object. The private parts of `T` would be defined in the private implementation of the package, by revealing the relationship between `T` and `Public` as follows[4]:

```
REVEAL
   T = Public BRANDED “Some brand” OBJECT ... END;
```

Now, let us look at how this prototypical object declaration is changed to define the `TrackerPosition` Shared Object. Recall that the Shared Object code generator must override all of the methods of an object in the generated code: these overrides are used to enforce the Shared Object semantics. Therefore, the code generator must define `T`, the top-level object used by the programmer, which prevents the programmer from using the “`T <: Public`” relationship to define the private parts of the object. Consequently, the programmer must create an additional opaque subtype for that purpose. Furthermore, since the base type of the object must be the shared object type `SharedObj.T`, the above prototypical declarations become:

4. The `BRANDED` keyword tells the Modula-3 type system that this is a unique type, even if there is another type in the program with the same structure (unlike its ancestor Pascal, Modula-3 uses structural type equivalence). The type system requires that one and only one revelation of an opaque type be branded. The BRANDED keyword can be optionally followed by a string (“Some brand” in this case) that names the branded type, otherwise a unique string will be supplied by the compiler. Brand names must be unique across the entire program.

```
TYPE
   T <: S;
   S <: Public;
   Public = SharedObj.T OBJECT ... END;
```

The proper use of the package also requires that the types used in the main interface do not reveal the internal data fields of the object. As above, `Public` would contain those parts of the object that the package designer wished the user of their package to see, and the internal details of `T` would be defined in the private implementation of the package, by revealing the relationship between `S` and `Public`, as follows:

```
REVEAL
   S = Public OBJECT ... END;
```

Finally, when an object is defined, the interface must be annotated with the `<*SHARED UPDATE METHODS*>` pragma to inform the code generator which of the object methods are update methods. For the `TrackerPosition` object, the pragma is:

```
<*SHARED UPDATE METHODS T.set, T.init *>
```

Given these declarations, collected in Figure 3-4, the code generator would define the object `T` in the generated code using the following revelation:

```
REVEAL
   TrackerPosition.T = TrackerPosition.S BRANDED
      TrackerPosition.Brand OBJECT ... END;
```

Six files are created by the code generator, the code for which can be found in Appendix A. `TrackerPositionCB.i3` and `TrackerPositionCB.m3` (Sections

A.2 and A.3) implement the callback object used for notification of changes to an instance of the Shared Object. `TrackerPositionSO.m3` (Section A.1) contains the implementation of the `TrackerPosition.T` object, including the dispatch function (`ApplyUpdate_T`) that processes incoming updates by relaying them to the individual dispatch stubs (`Stub_*`). The file also contains the method wrappers (`Shared_*`) that enforce the Shared Object semantics. Automatically generated code in these stubs and wrappers takes care of marshalling and unmarshalling arbitrarily complex method arguments and return values (such as large recursive data structures) between heterogeneous machines across the network.

In Modula-3, the process of marshalling data is known as *pickling* and is provided by the Pickle module. The `TrackerPositionSO.m3` file contains the necessary pickling routines used to copy the object state between sites. These routines do much of the work required to set up the synchronization protocol when new objects are copied to a process, and take care of ensuring that only one copy of an object exists in any given process. They also support the ability for a programmer to define their own routines to read and write the object data: the default set of routines simply pickles all the internal data fields, as discussed in Section 3.4.1.3. The `TrackerPositionPickle.i3` (Section A.6) interface is used to define these custom pickling routines. Finally, `TrackerPositionProxy.i3` and `TrackerPositionCBProxy.i3` (Sections A.4 and A.5) contain the proxy objects used to embed the Shared Object in an interpreted language such as Repo.

3.4.1.2 Callback Object Usage

The declaration of the Callback Object `TrackerPositionCB.T` is shown in Figure 3-6(a). To use the Callback Object, the programmer must declare a subtype of it that overrides the appropriate methods with procedures that perform whatever action is desired to handle notification of that update. A simplified version of such an object is shown in Figure 3-6(b). This subtype would be used as follows:

```
INTERFACE TrackerPositionCB;

IMPORT Tracker, SharedObj, TrackerPosition;

TYPE
  T <: PublicT;
  PublicT = SharedObj.Callback OBJECT
    METHODS
      init (obj: TrackerPosition.T): T;
      cancel ();
      pre_anyChange (READONLY obj: TrackerPosition.T);
      post_anyChange (READONLY obj: TrackerPosition.T);
      pre_init (READONLY obj: TrackerPosition.T): BOOLEAN;
      post_init (READONLY obj: TrackerPosition.T): BOOLEAN;
      pre_set (READONLY obj: TrackerPosition.T;
               READONLY val: Tracker.Report): BOOLEAN;
      post_set (READONLY obj: TrackerPosition.T;
                READONLY val: Tracker.Report): BOOLEAN;
    END;

END TrackerPositionCB.
```

(a) The generated interface, `TrackerPositionCB.i3`, containing the definition of the TrackerPosition Callback Object.

```
TYPE
   Callback = TrackerPositionCB.T BRANDED "My Callback" OBJECT
   OVERRIDES
     pre_set := Pre_set;
     post_anyChange := Post_anyChange;
   END;

PROCEDURE Pre_set (self: Callback;
                   READONLY obj: TrackerPosition.T;
                   READONLY val: Tracker.Report): BOOLEAN =
   BEGIN
     (* do something right before “set()” is called *)
   END Pre_set;

PROCEDURE Post_anyChange (self: Callback;
                   READONLY obj: TrackerPosition.T) =
   BEGIN
     (* do something right after “set()” or “init()” are called *)
   END Post_anyChange;
```

(b) An example Callback Object.

Figure 3-6: The `TrackerPositionCB.T` Callback Object. (a) contains the automatically generated interface to the Callback Object for `TrackerPosition.T`, and (b) shows a simplified example of the Callback Object in use. In this example, the programmer would insert the code to be executed just before the `set()` method is called in the body of the `Pre_set()` procedure. Similarly, the code to be executed after any update method would be inserted in the body of the `Post_anyChange()` procedure.

```
VAR  cbObj: Callback;
     trackerPos: TrackerPosition.T;
...
   cbObj := NEW(Callback).init(trackerPos);
...
```

In this simple example, after `cbObj` is initialized, all updates to `trackerPos` will result in the appropriate methods of `cbObj` being called. Notification stops when the object is garbage collected, or when notification is explicitly cancelled:

```
cbObj.cancel();
```

3.4.1.3 Passing State Between Processes

When a reference to a Shared Object is passed between processes, the Shared Object Runtime must copy the current state of the object to this new process. As discussed in Section 3.4.1.1, the Shared Object code generator defines the necessary routines to do this, and also provides a facility for programmers to define their own routines to read and write the internal state. The generated code for this example is shown in Figure 3-7. To redefine what is done to copy an object between sites, the programmer creates a subtype of the appropriate `Special` variable (defined in the generated `TrackerPositionPickler.i3`, shown in Figure 3-7(a)), overriding the `read` and `write` routines with the actions they desire. The routines are registered with the Shared Object Runtime using the `RegisterSpecial` routine (defined in the same interface). The default definition, extracted from the `TrackerPositionSO.m3` file, is shown in Figure 3-7(b).

The ability to define exactly what code is executed to copy the state between processes is important for both efficiency and usability reasons. The efficiency concerns usually focus on saving network bandwidth. For example, with some objects it might be possible to copy a small portion of the state and recreate the rest at the remote site. While

```
INTERFACE TrackerPositionPickle;
IMPORT SharedObj;

TYPE
   TSpecial <: SharedObj.Special;

PROCEDURE RegisterSpecial_T(sp: TSpecial);
END TrackerPositionPickle.
```

(a) The generated interface, `TrackerPositionPicler.i3`, containing the definition of the TrackerPosition Pickle Object `TSpecial`.

```
REVEAL
   TSpecial = SharedObj.Special BRANDED
                  "TrackerPosition.TSpecial"
   OBJECT OVERRIDES
     write := DefaultSpWrite_T;
     read := DefaultSpRead_T;
   END;

PROCEDURE DefaultSpWrite_T (<*UNUSED*>self: TSpecial;
                            shobj: SharedObj.T;
                            out: Pickle.Writer)
     RAISES {Pickle.Error, Wr.Failure, Thread.Alerted} =
   VAR
     obj := NARROW(shobj, S);
   BEGIN
     PickleStubs.OutRef(out, obj.data);
   END DefaultSpWrite_T;

PROCEDURE DefaultSpRead_T (<*UNUSED*>self: TSpecial;
                            shobj: SharedObj.T;
                            in: Pickle.Reader)
   RAISES {Pickle.Error, Rd.EndOfFile, Rd.Failure,
           Thread.Alerted} =
   VAR
     obj := NARROW(shobj, S);
   BEGIN
     obj.data := PickleStubs.InRef(in, TYPECODE(Tracker.Report));
   END DefaultSpRead_T;
```

(b) The default Pickle Object for `TrackerPosition.T`, taken from the `TrackerPositionSO.m3` file.

Figure 3-7: The default `TrackerPosition.T` marshalling code. The Shared Object system must know what to copy when objects are passed between machines. The code generator provides a default set of reading and writing routines that copy the state of an object. In this case, the internal `data` field is copied.

the system would function without this facility if efficiency was the only concern, there are times when the ability to define exactly what happens when objects are passed between processes is necessary.

Typically, this happens when the object contains data fields that can not, or should not, be copied between processes (see Section 3.4.3); for example, if an object contained pointers to data structures created by the operating system or external libraries (such as OpenGL display lists, or Windows NT/X11 window handles). These objects cannot be copied because they are meaningless in the destination process. Instead, the information needed to recreate them would be passed to the new process, where the appropriate operating system or external library routines could be called to recreate the data. Another example of the need for custom picklers is when an object contains state that is (conceptually) local to each process. This state should not be copied, but rather initialized to some default values in the destination process. An example of this will be seen in the implementation of Repo-3D in Section 5.5.

3.4.1.4 Additional Tracker Examples

In the previous sections, we presented a detailed example of the creation of a Shared Object, taken from Coterie's Tracker package. While this object itself is extremely simple, more advanced tracker objects can be implemented using it. For example, assume a tracker is being handled by some process, and its reports are distributed via a `TrackerPosition` object, which we will call `tobj`. Now, suppose a client wants to receive at most one tracker report per second. A second `TrackerPosition.T`, called `slowobj` could be created in the process reading the tracker device. It would be updated less often by associating a simple callback object with `tobj`, whose `post_set` method is overridden to update `slowobj` at most once per second (recall that the `post_set` method is called just after the associated object's *set* method is called, as discussed in Section 3.3.3). The Modula-3 code implementing the callback object, called `LowFreqTracker`, is shown in Figure 3-8(a), and code showing how a programmer would use it to create a low frequency `TrackerPosition.T` object is shown in Figure 3-8(b). Local copies of `slowobj` can now be obtained by clients who want to have updates sent to them less frequently. Notice that this approach allows great flexibility. For example, the process in which `slowobj` is created determines where the filtering is done; in this case it is being

```
TYPE LowFreqTracker = TrackerPositionCB.T OBJECT
     slow: TrackerPosition.T;
     interval, next: REAL := 0.0;
   OVERRIDES
     post_set := LFSet;
   END;

PROCEDURE LFSet(READONLY obj: TrackerPosition.T;
            READONLY val: Data): BOOLEAN =
   BEGIN
     IF self.next < Time.Now() THEN
        self.slow.set(val);
        INC(self.next, interval);
     END;
   END LFSet;
```

(a) A `TrackerPositionCB` used to create a slowly changing `TrackerPosition.T`

```
   (* assume tobj is the normal tracker object *)
   slowobj := NEW(TrackerPosition.T).init();

   slowcb := NEW(LowFreqTracker,
                 slow := slowobj,
                 interval := 1.0).init(tobj);
```

(b) The new callback, `LowFreqTracker`, is used to create the new `TrackerPosition.T`

Figure 3-8: A low frequency tracker object. (a) A simple use of the `TrackerPositionCB.T` callback object from Figure 3-6(a), used to create a variation of a tracker that is updated at most once a second. Each time the `post_set` method is called, it checks to see if the time interval has passed. If it has, it updates the `slow` object and increments the `next` field to wait for the next interval to expire. (b) These two lines of code allocate a new `TrackerPosition.T` object and the `LowFreqTracker` callback object that updates it based on `tobj`, the full frequency `TrackerPosition.T` object.

done in the local processes, but it could just as easily be done in the remote process or some other process in between.

3.4.2 The Shared Object Runtime

There are a number of aspects of the Shared Object runtime that are interesting, especially with respect to the lessons learned during its development. Among other things, the Shared Object runtime is responsible for ensuring that there is one and only one instance of any Shared Object in a process, performing distributed garbage collection, maintaining

a set of threads to process incoming events, sequencing updates (if the process is the sequencer for the object in question), and forwarding update requests to the sequencer (if the process is not that object's sequencer).

We ensure there is one and only one instance of a Shared Object in any process primarily for efficiency reasons: multiple copies would waste memory, as well as require that all update methods would have to be executed multiple times (once for each copy of the object). Furthermore, allowing multiple copies would complicate the runtime by requiring it to keep track of multiple copies of an object in each process. To ensure that there is only one copy, the runtime maintains a table of all the replicated objects in a process, indexed by the unique global object identifiers. By using the pickling facilities discussed in Sections 3.4.1.1 and 3.4.1.3, the runtime checks for the prior existence of an object whenever it is passed to a process: if the object already exists in the destination, the current one is used, otherwise a new object is created and entered into this table. This table is also used to dispatch incoming updates to the local objects.

We take further advantage of this check for the prior existence of a Shared Object in a process to improve network bandwidth utilization by only sending the global object identifier when a Shared Object is embedded in an argument to an update method, rather than pickling the entire object state. Therefore, if the object already exists in that process, the existing copy can be used without having needlessly copied the object state across the network. If the object does not exist at the destination site, the Shared Object runtime makes a remote method call to its sequencer to obtain the object, which may in turn require the sequencer to make remote method calls to other sequencers, or to one of its clients to obtain the object. These additional network accesses may increase the time taken to pass objects between processes, but can result in a significant savings when an object already exists in the destination processes, especially when the object state is large.

Distributed garbage collection takes advantage of the *weak reference* facility provided by the Modula-3 runtime. A weak reference is a reference to an object that the Modula-3 garbage collector does not consider when determining if an object is referenced in a process. When a weak reference is created, a cleanup routine can be provided; when there are no longer any non-weak references to an object, all weak reference cleanup routines

are called just prior to the object being garbage collected. The Shared Object system only maintains weak references to the local objects in its object table (mentioned in the previous paragraph). When all real references to a Shared Object are removed from a process, the runtime notifies the sequencer that this process no longer has a copy of the object, allowing the sequencer to keep track of where real copies of an object are and to stop routing updates to those processes that no longer have copies of an object.

In the next two subsections, we will discuss two issues in greater depth: the management of threads by the runtime, and the handling of exceptions and return values in update methods.

3.4.2.1 Thread Management

Figure 3-3 shows a simplified representation of the data flow through the system. Each process maintains a connection to its sequencer, represented as an Event Port in the diagram. If a process is a sequencer, it would also maintain a connection to each of the clients in its cluster. All Shared Object communication is performed through these communication channels. As shown in the diagram, each port uses two threads to process events read from, and written to, the Event Port. While this may seem like an excessive use of threads, especially for a sequencer that may be talking to a large number of peers, it results in a cleaner, more robust, and more efficient implementation. One of the lessons learned from this implementation is that a judicious use of multi-threading increases the responsiveness and robustness of the system. Since these lessons were learned during the development process, no quantitative results are available to illustrate them.

This lesson is especially true when deciding how to handle incoming update events. To apply a sequenced update to an object, the following actions must be taken:

1. An exclusive write lock is acquired for the object.
2. All "`pre_`" methods are called for each Callback Object associated with the object.
3. The update method is executed.
4. All "`post_`" methods are called for each Callback Object associated with the object.
5. The write lock is released.

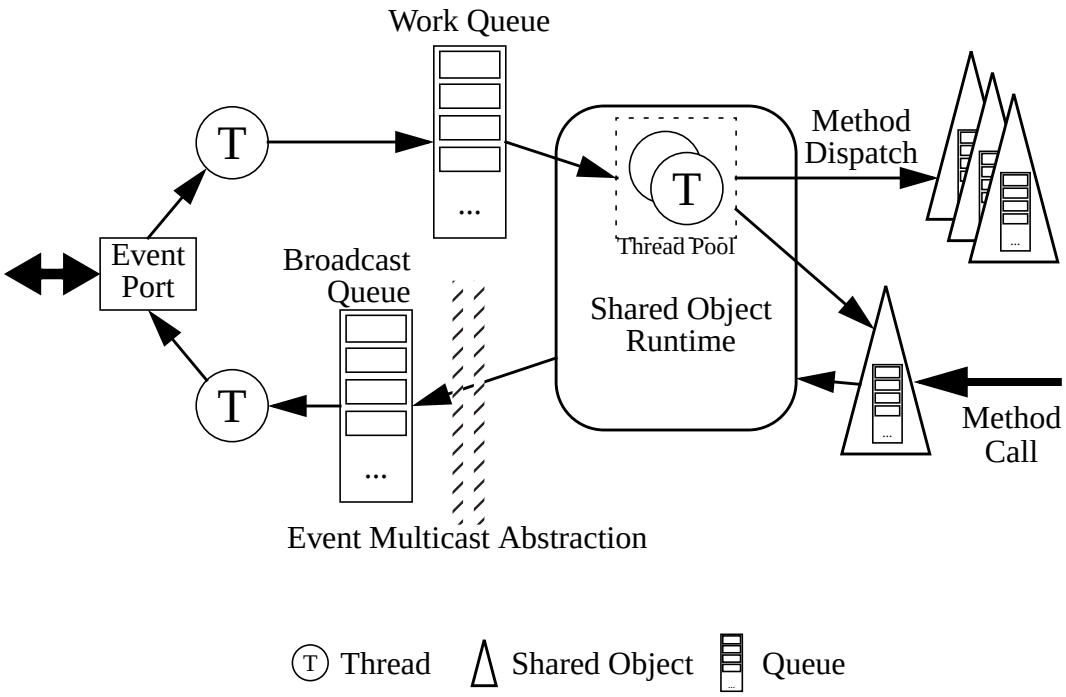


Figure 3-9: Data Flow in the Shared Object System. The Shared Object package is built on top of an Event package that presents both unicast and multicast event distribution abstractions. (Currently, the event layer is implemented with TCP sockets.) Each Event Port in a process uses two threads, one to read events from the port and add them to an event queue, the other to read events from a queue and write them to the network. The Shared Object package maintains a Thread Pool to process incoming events. Each Shared Object maintains a queue of out-of-sequence events that have arrived, but cannot yet be processed. An update method call may require a message to be sent to the network, and wait for a reply to unblock it and allow it to continue.

Unfortunately, of these steps, only the last one is guaranteed to happen in a timely manner. If there are readers accessing the object, step 1 will block until they exit. While this will usually result in at most a short delay, steps 2–4 could take a significant amount of time. Even though programmers are encouraged not to write update methods or Callback methods that do substantial amounts of work, there is no way to ensure that this does not happen. Therefore, multiple threads must be used to ensure incoming updates are processed in a timely manner.

However, too many threads results in too much parallelism, which is also undesirable, as the process will then spend substantial amounts of time context switching between these threads. To provide some control over this parallelism, we implemented a Thread

Pool abstraction for use in processing incoming events[5]. When a Thread Pool is created, the programmer specifies the maximum number of running threads, the maximum number of idle threads and a queue to obtain "work" from. In this case, the work queue is the incoming event queue, as shown in Figure 3-9. The application programmer can interact with this Thread Pool in two ways, to tune the performance of the system:

- The programmer can cause a thread to block until all the queued Shared Object events are processed. This allows more processor time to be devoted to incoming updates, and is needed primarily because Modula-3 threads are all executed at the same priority. Ideally, higher priority threads would handle incoming updates so that the system would guarantee all replicated objects get updated in a timely manner
- If a method is known to take a significant amount of time (i.e. blocking on I/O, accessing a network service, etc.), the current thread can be removed from the Thread Pool. The current thread will no longer be included in the Thread Pool's thread count, allowing an additional thread to be created if necessary.

This latter facility is needed because we want Callback Object methods to be able to call update methods of other Shared Objects. Since we want Callback methods to be executed synchronously with the corresponding update methods, rather than being added to a queue for later processing, the Callback Objects are executed by the same thread that executes the update method. When the update is being executed by a thread from the Thread Pool, this thread *must* be released from the pool, or the system may deadlock, as follows:

- An update method is called and a message is sent to the sequencer to put that update in sequence.
- The thread invoking the method blocks until the sequencer returns the sequenced update to the process.
- The number of threads in the Thread Pool handling incoming messages is limited, so if all of those threads block by calling update methods, the system will not be able to han-

5. The Thread Pool abstraction turned out to be so useful, we eventually exposed it into Repo, as will be seen in the examples in Chapter 4.

dle replies from the sequencer to unblock those threads, and the system will be deadlocked.

Therefore, the runtime automatically removes any thread from the thread pool before it blocks waiting for an update to be sequenced. The alternative would be to allow an unlimited number of threads to handle incoming Shared Object events, but this is a bad idea: if the system could not handle events as fast as they arrived, a progressively larger number of threads would be created. This would slow the system down further, causing more threads to be created, and so on until resources were exhausted.

3.4.2.2 Exception and Return Value Handling

Since update methods are executed on every replica of an object, the Shared Object runtime must deal with the values returned from, and exceptions raised by, those methods. No special handling is required for non-update methods: when a non-update method returns a value or raises an exception, it is simply passed on to the calling method, as would be expected. Similarly, in the process that initiates an update method, return values and exceptions are also passed back to the calling method; since the runtime arranges to have the thread that initiated the call execute the method when the sequence number is returned from the sequencer, the exceptions will propagate through the call stack in the normal manner.

When a value is returned by an update method in a process aside from the one that initiated the update, it is simply ignored. Similarly, for the vast majority of object update methods, if an exception is raised in the initiating process, it will be raised when the method is executed in all replicas. In this case, since the exception is passed on to the caller in the process that initiated the update, it can be safely ignored in all other processes because the initiating process can take whatever action is necessary to deal with the exceptional condition; therefore, the runtime catches these exceptions and ignores them in all processes aside from the initiating process.

There are cases, however, when an exception will only be raised in some subset of the replicas. For example, if a method needs to acquire local resources (such as memory, or space in a local file system), the request for that resource may fail in some replicas, but

not in others. If the update succeeds in the process that initiated it, the calling thread has no way of knowing that the update failed in one of the replicas because it does not receive an exception; update distribution is asynchronous, and no information is returned to the process that invokes an update method regarding the invocation of that method in other processes. The Shared Object system requires that the programmer decide which exceptions will be raised in all replicas (i.e., those that depend only on the global object state) and which might be raised in a subset of the replicas. To help the programmer deal with these latter situations, the Shared Object package provides the programmer with a special exception called `SharedObj.Fatal`. If a programmer raises this exception in a method, the Shared Object runtime assumes that something has failed in this replica that may not have failed in all others, and marks this replica as invalid.

While the process that issued the update has no way of knowing that one or more of the other replicas may now be invalid, any attempt to access an invalid replica, or to pass it to another process, will fail with a `SharedObj.Error` exception. How to proceed is then up to the process containing the invalid replica. While this is not an ideal solution, it is sufficient for our needs. For example, the process could attempt to acquire a new copy of the object, or follow any other reasonable course of action. The runtime does not attempt to acquire a new copy of the object for the programmer because whatever resource was unavailable when the update method failed will probably still be unavailable, and programmer intervention will be required to properly handle this situation.

3.4.3 Restrictions

As with Network Objects, there are restrictions on what Modula-3 types are valid for use as arguments to method calls. For Shared Objects, the restrictions arise from the need to package arguments to update methods into the update messages distributed to all replicas. Therefore, no data value can be used that is specifically associated with one process (e.g., a thread or a condition variable) or that has state that cannot be accessed repeatedly with consistent results (e.g., a file reader or writer).

While Network and Shared Objects are allowed to be used as arguments to update methods (so that data structures such as distributed lists can be constructed), these objects

Benchmark	Time (milliseconds)
Simple Access (read or write)	0.03
Replicated Read	0.05

Table 3-2: Local method call performance. A comparison of performance for replicated object read methods for synchronized (Replicated) and unsynchronized (Simple) objects. Times are an average of 10 successive calls of the given type.

should never be accessed from within update methods, as the results are both inefficient and unpredictable. Accessing Network Objects from an update method is inefficient because each of the replicas will need to perform the same remote access to the Network Object. Accessing both Shared and Network Objects from an update method can result in unpredictable behavior, as the object being accessed may change between the accesses, and therefore return different information to some of the replicas, potentially causing the replicas to become inconsistent.

3.5 Performance and Usability

While performance was not the primary concern when designing and building the Shared Object package, it was definitely a factor in our design. Usability, on the other hand, was a significant concern, and was the primary motivation for tightly integrating data replication with a programming language. While we have not done extensive performance or usability analysis, we would like to show two things in this section. First, that the performance of the system falls within our expectations, and is acceptable for the kinds of applications for which we designed Coterie. Second, based on the similarities between our object system and that of Orca, we will argue that the recent usability and performance analyses of Orca [Wilson and Bal, 1996, Bal et al., 1998] justify the approach taken for Coterie, namely to implement replicated data using objects based on a write-update protocol with function shipping and totally ordered group communication.

3.5.1 Shared Object Performance

We measured the performance of the Shared Object system, the results of which are shown in Tables 3-2 and 3-3. The tests were performed between a group of Sun Ultra2

Benchmark	Time (milliseconds)
Remote Access (read or write)	1.04
Replicated Write (2 clients)	4.57
Replicated Write (4 clients)	4.75
Replicated Write (8 clients)	8.2

Table 3-3: Distributed method call performance. A comparison of performance for network access methods for each of client-server (Remote) and synchronized (Replicated) object calls. Times are an average of 10 successive calls of the given type. For the Replicated Writes, the time taken is shown for 2, 4 and 8 clients of a single sequencer.

machines, connected via a 10baseT ethernet hub, and were implemented in Repo (Chapter 4). For each of the six tests, 10 calls of the given type were performed and the average time for a single call obtained as an average over total time. This is the worst performance situation a Coterie programmer would typically encounter: Repo has more overhead per call than Modula-3, and a 10Mb/sec hub is the slowest network configuration that would be used for the system. We use this configuration, however, because Coterie programmers typically build replicated objects in Repo, not in Modula-3. Therefore, the slower, hubbed network limits the scalability (as seen in the last measurement in Table 3-3) rather than the speed of individual calls.

Each call sets a data field in the corresponding object to an integer value. The Simple measurement is a call to a local, unsynchronized object, and represents the minimum time needed to make a Repo method call. The Remote measurement is for a client-server access to a second process, and represents the best time we could hope to achieve for a round trip message, since the Network object package has been optimized for the case of repeated method calls between processes (TCP connections are cached, so each subsequent call typically reuses an existing connection). The Replicated measurements correspond to read and write access to synchronized, replicated objects, which are implemented by the Shared Object package.

As can be seen, the Shared Object package achieves its primary goal of fast local read access to the replicated data, as the reads are only slightly slower than reads of non-synchronized objects. The price to be paid for these fast reads is significantly slower writes: the Shared Object package is currently over four times slower than the Network

Benchmark (milliseconds)	Fast Ethernet	Myrinet
Remote Access (read or write)	0.242	0.0406
Replicated Write (8 clients)	0.244	0.0647
Replicated Write (32 clients)	0.385	0.0847

Table 3-4: Orca Method call performance. A comparison of latencies introduced in object access for remote and replicated objects. The measurements are taken from Table II in [Bal et al., 1998].

Object package when updating a Shared Object, for 2 or 4 replicas. The performance drops off substantially after that. There are a variety of reasons for this poor performance, all of which have to do with the simplistic nature of the current implementation. For example, with a large number of clients, our use of TCP (instead of multicast UDP) causes a significant increase in network activity, and therefore collisions, on a shared ethernet. Even on a switched network, the sequencer must send the update to each client in sequence, which causes significant performance degradation. Other aspects of the implementation hurt the performance and could also be improved, such as the user-level thread performance of Modula-3. However, since the system performs well enough for our applications, this was deemed a low priority. An optimized implementation, on fast networks and computers, should be able to achieve update performance (for a small number of clients) similar to the Remote Access numbers.

This is exactly the kind of performance obtained by the Orca system, as shown by the performance analysis recently done by its designer [Bal et al., 1998]. Table 3-4 contains their measurements, taken from Table II in the paper. The authors measured the performance of their system on a network of 200Mhz Pentium Pro machines, connected by either Fast Ethernet or Myrinet (a 1.28Gb/sec network with extremely low latency). As can be seen, their performance numbers for Remote Access are approximately 1/5th of ours, and their Replicated Write times are in line with the Remote Access ones. Orca uses three different protocols for distributing updates, all of which rely on efficient multicast. The protocol used for a given update is chosen dynamically to optimize network performance, based on the size of the update message and other factors. The Myrinet performance is further optimized by having the sequencers implemented in the Myrinet network switches. Furthermore, Orca is a compiled language and optimizes its runtime at multiple

levels, from the operating system up to the application. This contrasts sharply with our simple, TCP-based implementation of the Shared Object runtime system.[6]

However, the performance measurements for individual calls are not the focus of [Bal et al., 1998]. Instead, they focus on a quantitative performance analysis of 10 applications implemented in Orca, and a comparison of Orca to two other DSM systems: TreadMarks, an optimized page-based DSM [Keleher et al., 1994], and CRL, an object-based DSM that uses a directory-based write-invalidation scheme [Johnson et al., 1995]. The latter analysis demonstrates that for a wide range of applications, Orca's approach to DSM is superior to either of the other two. This is an important finding, because common wisdom typically holds that write-update protocols are inefficient.The reason Orca defies this common wisdom, and also why it performs well for the ten applications analyzed in the first part of the paper, turns out to be its intelligent approach to object placement.

Like Coterie, and unlike most DSM or DOM systems, Orca uses a combination of replication and single-site client-server objects. In Orca, objects that have a high read/write ratio are replicated, and those with a low read/write ratio are migrated to the process that is updating them and accessed remotely by other processes. Bal observed, as we also did, that it is these latter objects, with a low read/write ratio, that are responsible for the typical poor performance of write-update protocols. However, when replicated objects are read a least once in each process between writes (which is what Bal considers to be a high read/write ratio), a write-update protocol performs quite well, for two reasons: first, it is more efficient to send an update to a process, instead of just an invalidation message, if that process is going to request the updated object before it is changed again, and second, if many processes are going to request a copy of the updated object, an update protocol will allow the update to be broadcast to all processes at once, instead of unicast to each host as they request it.

Interestingly, the authors also found that for their applications, the best performance was achieved when programmers manually chose which objects were replicated

---

6. The runtime is designed to support a more efficient implementation, if it becomes necessary, without necessitating any visible changes at the programming layer.

and which were to be accessed via client-server semantics (and where those objects should be located), rather than having the system make those choices dynamically. While the Orca runtime system will make decisions similar to those made by the programmers regarding object placement and replication, it invariably makes incorrect initial choices that must subsequently be corrected. They point out that since these corrections are typically made relatively quickly, the difference is typically insignificant for a long running program, and they prefer to have Orca make the choices to free programmers from such decisions.

The results presented in this paper are relevant to our work because of the similarity between Coterie's combination of Shared (replicated) and Network (client-server) Objects and Orca's object system. The point that manual placement is more efficient, if slightly less convenient, is encouraging because a programmer must make these manual decisions in Modula-3 when they define an object: an object is either a Network or Shared Object based on the class it inherits from, and there is no facility to convert between the two (either manually or automatically). Bal and his colleagues point out that in the typical case, it is desirable to have the system make the placement decisions, since the idea of DSM is to hide the details of distribution from the programmer, and the optimal choice may be machine-dependent. We agree with them, but do not provide this facility because of the implementation overhead. We also decided early on to keep the Network and Shared Object packages separate since the Network Object package is already widely used: supporting conversion between the two would be difficult without integrating them.

It might seem that a write-update protocol based on total global ordering of updates via a centralized sequencer would be inefficient because all updates must be broadcast to all replicas, and that they must pass through a centralized sequencer (which would be a potential bottleneck). However, the analysis by Bal shows that this criticism is simply unfounded when replicated objects are used in concert with client-server ones, as we have done in Coterie. First, when objects with a high read/write ratio are replicated, a write-update protocol turns out to be more efficient than any other protocol, as described above (primarily because the objects would end up being refreshed after each write anyway, but the systems would not be able to take advantage of network broadcasting to dis-

tribute those updates). By using client-server semantics for objects with a low read/write ratio, we avoid sending around frequent updates that will not be read. Second, because update operations are done at a high level (the level of the method call, instead of the level of memory access), a single update method can encapsulate a set of updates to the object, reducing the number of updates to be distributed in practice. Finally, there will only be a possibility of contention at the centralized sequencer when objects have many, frequent updates applied to them. In this case, however, the objects can be changed to client-server objects because their read/write ratio will probably be low.

Other performance issues with Orca have been discussed in the context of an analysis Wilson did of its usability [Wilson and Bal, 1996] (we will return to the usability issues raised by this paper in the next section). Chief among the issues, and one that we have also experienced, is the slowdown incurred by always blocking the issuer of an update message for the duration of the round trip to the sequencer. Their proposed solutions focus on figuring out how to pipeline multiple update calls at the compiler level, without violating the serializability of the model. It is not clear how to solve this problem without doing code analysis, which is prohibitive to do at the level of a language add-on such as the Shared Object package.

Wilson also identified a number of performance problems with Orca that have been addressed to a certain extent by our design. One performance problem their students complained about was the inefficiency of unsynchronized replicated objects. While Orca supports defining objects as unsynchronized (they call them *non-shared*), these objects go through the same protocols as synchronized ones, making them too inefficient for use in simple sequential programs. Coterie does not suffer from this problem because, since the Shared Object package is a tightly integrated extension of Modula-3, programmers are free to use normal Modula-3 data objects.

Another performance problem encountered while using Orca arises from its strict enforcement of the dictum that internal data can only be accessed via methods. Even when performance becomes a problem and a programmer is sure they can safely access the internal data fields without violating the model, there is no facility for them to do this. In the Shared Object system, on the other hand, the internal interface that exposes an object’s

internal structure (e.g., Figure 3-5(a)) can be made public and used by programmers if they need to. It is up to the programmer to ensure they know what they are doing, so this facility is not to be used lightly. We make use of this ability in the implementation of Repo-3D (see Section 5.5).

3.5.2 Shared Object Usability

While we have provided well-reasoned justifications for the design of the Shared Object package, we have not done any formal experimentation to measure the package's ease-of-use as a distributed programming system for exploratory programming, except anecdotally. More importantly, most of the researchers who have built applications using Coterie have never ventured into Modula-3 or used the Shared Object package directly, instead confining themselves entirely to Repo. The two major programming tasks that have been undertaken using the Shared Object package are the implementations of Repo and Repo-3D, and both of these were done by the author. While we can attest to the ease with which these packages were created using the Shared Object system, the experiences of system designers using their own system should be taken with a grain of salt.

More convincing support for the hypothesis that this is a useful programming model can be found by again turning to Orca, this time looking at an analysis Wilson did of its usability [Wilson and Bal, 1996]. As with the performance analysis in [Bal et al., 1998], this usability analysis was published after our work was completed, so that we could not take advantage of their experiences when we designed the Shared Object package. However, the similarity of our experiences is compelling.

Wilson proposed a suite of medium-sized, realistic applications that can be used together to evaluate the usability of parallel and distributed languages and systems, called the Cowichan[7] problems [Wilson, 1994]. These problems are intended to determine the usability of programming systems for writing efficient parallel programs. In contrast, the

7. The problem set is named after a place and tribal name from the Canadian Northwest, and was chosen by Wilson to acknowledge the debt his work owed to the Salishan Problems of Feo [Feo, 1992], which is also a tribal name and the name of the conference center in Oregon where the Salishan Problems were formulated.

benchmarks typically done by system developers, and supported by other suites, tend to assess only the performance of programming systems (or hardware), and not the usability. The suite was selected to cover a wide spectrum of application domains and parallel programming idioms.

To assess the usability of Orca, the authors had six different students each implement one of the Cowichan problems. They first implemented a sequential version in C to familiarize themselves with the problem, and then implemented a parallel version in Orca. What is interesting is that the lessons and experiences related by the authors are quite similar to the experiences related to us by those who have been using Coterie, although there are some we had not discovered because of the nature of our application domain. For example, while we had discovered that the atomicity of objects is a problem in certain cases, we had not encountered problems based on the inability to partition objects. Wilson found that this was a problem for some of the numerical problems in the suite, which would benefit from partitioning matrices over multiple sites. To overcome this problem, programmers had to partition the objects themselves and use the shared objects for communicating changes: in this case, the programming model effectively drops down to a form of message passing.

While Coterie shares this problem with Orca, the fact that the model supports a fall-through to a style of programming that is best described as “structured message passing” supports our claims about its flexibility. While this approach may not be transparent to the programmer, implementing a kind of message passing by communicating via Shared Objects takes only a few lines of code (in Repo) and allows the programmer to “get the job done,” which is exactly what is needed for exploratory programming.

The other major problem that Wilson, and others, found with Orca is the inability to apply an operation to multiple objects atomically. In contrast to the partitioning problem, this is something we have noticed with Coterie. There are a number of approaches to dealing with this issue that we have considered, but none is particularly clean, efficient or easy to implement. We will return to this issue in Chapter 7 when we discuss future research directions.

One final usability problem we have encountered with Coterie is caused by the synchronous nature of the Callback Objects. While synchronous execution of the methods of the Callback Objects is often necessary to guarantee that the Shared Object is in a known state, or (in the case of the `pre_` methods) to be able to access the state just before the update is applied, asynchronous notification of updates is often sufficient. More importantly, it is often necessary: since the Callback Object methods and the update methods are currently executed synchronously by the same thread, programmers are not supposed to do any significant work in the Callback Object methods as access to the object is blocked while the methods are being executed. Since programmers often wish to execute arbitrary actions from the Callback Object methods, they often end up building their own asynchronous event notification queues in Repo. It would be useful if the system were to support asynchronous notification directly.

Despite the problems raised, Wilson and Bal conclude that Orca is a useful and easy to learn parallel programming environment. They found that, while programmers do have to concern themselves with communication and synchronization, the language and model encourage them to think about these issues in a highly structured manner. This corresponds to our anecdotal findings: by putting distribution concerns into the object design, programmers deal with it once in a structured way, and then spend the rest of their development time dealing with application development without concerning themselves too much with distribution.

Therefore, based on both the similarities in the programming model of the Shared Objects and Orca, and our experiences and lessons learned, it seems reasonable to draw similar conclusions about the Shared Object package. In both systems, programmers do occasionally have to concern themselves with communication and synchronization, but the language and model encourage them to think about these issues in a highly structured manner. This was one of our original motivations: prior to this work, we had observed that much of our programming time was being spent implementing communication and synchronization protocols, with a surprisingly small amount of time and code devoted to other aspects of the programs we were building. With the Shared Object package, the opposite is

now true. While the distribution issues can not (and should not) be ignored, they now occupy a relatively small amount of time and code.

3.6 Discussion

There are four factors behind the design of the Shared Object package: efficiency, simplicity, transparency, and flexibility. From the previous discussion, it should be apparent that the package satisfies our efficiency concerns, both in terms of the timely distribution of updates and fast local read access. We have also found that the package is simple to use. First, by following a few simple guidelines, listed at the beginning of Section 3.4.1, building replicated objects is a straightforward task. Second, once the objects are created, they can be used just like any other Modula-3 object. This allows the programmer to deal with the issues of data replication in one place (when the object is defined) and ignore distribution elsewhere, instead concentrating on other application details.

The Shared Object package also offers almost complete network transparency: once an object is defined, it can then be used the same way as any other Modula-3 object. The objects differ from normal Modula-3 objects in a few subtle ways: there is an implicit lock around all method access (which programmers only need to be aware of when implementing the object methods), and the methods may all raise the `SharedObj.Error` exception without the programmer raising it in any of the method bodies. This network transparency is especially important when the distribution semantics of an object must change, so a programmer needs to change the definition of an object from a Shared Object to a Network Object or to some other kind of object: aside from changing the type that the object in question inherits from, and perhaps making other changes to the implementation that are required based on its new usage, the effects of the change will be minimal (typically, the programmer will have to catch and handle the new `SharedObj.Error` exception).

The final factor in the design is our desire to create a system that is as flexible as possible, motivated by our focus on exploratory programming. The Shared Object package turns out to be very flexible because we define consistency in terms of method execution (both the order of execution and whether they modify the global state), but say almost

nothing about the contents of the objects data fields. For example, the programmer of an object has great flexibility in partitioning the work into parts executed once (at the calling site) and parts executed at all sites, by taking advantage of the fact that update methods are broadcast and executed at all sites, while read methods are not. Work can therefore be partitioned by having a read method call an update method after performing some work locally. This same technique can be used to lessen the impact of the restrictions on update method argument types, for example, by having a read method manipulate the restricted argument locally and use the results as arguments to an update method call.

We make use of the ability to perform arbitrary actions in methods in the implementation of both Repo and Repo-3D, but especially in Repo-3D. Since part of the state of each graphical object is global, and part is local to each machine (both the part that associates the conceptual graphical object state with the concrete state used by the rendering subsystem, and the local variations to the graphical state), we can manage these data structures in a straightforward and efficient manner by manipulating local data within the read methods and global data within the update methods. See Chapters 4 and 5 for more details on how we took advantage of the model in the implementation of those packages.

CHAPTER 4 Repo

"The life of a repo man is always intense" – Miller, from *Repo Man*

A commonly used approach to exploratory programming is to provide the programmer with an interpreted language with which they can build their applications. Interpreted languages offer two benefits for exploratory programming. First, they allow the programmer to avoid the compile-link cycle. Second, they allow programmers to incrementally (and interactively) develop and test applications. The former benefit is declining in importance as computers become faster and byte-compiled languages such as Java (that do not require applications to be linked into a single program) become more popular. The latter benefit, however, is significant, and is our primary motivation for using an interpreted language.

Unfortunately, when Coterie was being designed, there were no interpreted languages that satisfied our needs. While a number of interpreted languages provide simple client-server access to distributed data (e.g., Python [van Rossum, 1995] combined with ILU [Janssen et al., 1998], TCL-DP [Perham et al., 1997], and Obliq [Cardelli, 1995]), none support replicated data. Of these languages, Obliq has the most elegant and powerful model for distributed programming, relying on distributed lexical scoping as its key mechanism for managing distributed computation. For our purposes, Obliq also has the advantage of being implemented in Modula-3, and having its data distribution based on the Network Object package; as was pointed out in Chapter 2, the existence of Obliq was one of the factors that influenced our decision to use Modula-3 in the first place.

From the perspective of researchers developing interactive graphical applications, the major shortcoming with Obliq, as with Modula-3, is the lack of support for data replication: in Obliq, all data items (objects, arrays, and variables) have client-server distribution semantics. In Modula-3, we solved the problem by creating the Shared Object replicated data distribution package (the topic of Chapter 3) that, when combined with the

Network Object package, presents the programmer with a DOM programming model supporting both client-server and replicated data distribution semantics. Since we want to support one common programming model throughout the system, we uniformly extended the type system of Obliq so that all its data items can be distributed using synchronized and unsynchronized replicated distribution semantics, in addition to the client-server semantics already supported by the language. The resulting language is called Repo (*Rep*licated *O*bliq).

Unlike the Modula-3 DOM, in which only the programming language objects (and not other data items) are distributable with all three semantics, in Repo all data items (i.e., objects, arrays and variables) can take on any of the distribution semantics. In addition, Repo's objects are more general than Modula-3's since the object data fields are exposed and updates to them are distributed without the need to define update methods. As with the Modula-3 DOM, the objects can be mixed and matched in arbitrary ways, but because the distribution semantics extend across the entire type system, a wider range of interesting data structures can be developed. Repo also includes new libraries, and enhanced versions of a number of Obliq ones, that are needed to support exploratory programming in our domain. These include simple support for reflection, HTTP clients and servers, regular expressions and so on. By allowing distributed applications to be developed in a few lines of interpreted code, Repo turns out to be an excellent language for exploratory programming of distributed interactive applications.

In the rest of this chapter, we will describe Repo, often by contrasting it with Obliq. While we will provide enough information about Obliq that the reader can appreciate Repo's design, there are many aspects to Obliq that are not changed in Repo, and will therefore not be discussed in depth. For a more in depth discussion of Obliq, and examples of it in use, see [Cardelli, 1995]. The importance of Repo is both as an interpreted language supporting replicated data, and as an example of how a complex application can be built with the Shared Object package. We will discuss both of these topics in this chapter.

First, in Section 4.1, we will discuss other distributed interpreted languages. We will then turn our attention to the design of Repo, focusing on how it cleanly extends Obliq to support replicated data. An overview of Obliq and Repo will be presented in Sec-

tion 4.2, followed in Section 4.3 by a discussion of how support for replication in Repo changes the distributed semantics of Obliq. In Section 4.4, the syntax changes to Obliq object declarations to add support for replication will be discussed, and a new built-in module for controlling object replicas is described briefly in Section 4.5.

In Section 4.6, we will present a number of illustrative examples of Repo in use. Some interesting aspects of the implementation will be discussed in Section 4.7, including an overview of how the Shared Object package was used to implement Repo's replicated objects. Finally, Section 4.8 will close the chapter with a discussion of Repo's usability, based on discussions with the programmers in our lab who have been using it.

4.1 Related Work

There have been many interpreted procedural languages created over the years, and a number of them have supported, or been extended to support, client-server data distribution. For example, two of the most popular interpreted languages, Tcl [Ousterhout, 1990] and Python [van Rossum, 1995], include support for distribution via client-server semantics. Python supports CORBA compatible client-server distribution via ILU [Janssen et al., 1998], whereas a number of different extensions to Tcl have been implemented that support RPC-style distribution (e.g., [Nog et al., 1996] and Tcl-DP [Perham et al., 1997]). Unlike these language extensions, Obliq was designed from the start for distributed programming [Cardelli, 1995]. Obliq's model of computation is built around the use of lexical scoping and higher-order functions in a distributed context, as will be explained in Section 4.2. Unfortunately, Obliq supports only client-server data sharing.

We are interested in interpreted languages that present an object-oriented or procedural programming model, similar to that of Modula-3, including support for data replication. To our knowledge, no other such languages exist. There have been distributed interpreted languages that present the programmer with programing models that differ from the usual procedural style, especially in the Agents community (e.g., Telescript [White, 1994], and Agent Tcl [Gray, 1996]). However, these languages provide support for distributing computation through code mobility, and do not support building complex distributed applications needing efficient replicated data.

4.2 An Overview of Obliq and Repo

Obliq is a lexically-scoped, untyped, interpreted language for distributed object-oriented computation. It is implemented in, and tightly integrated with, Modula-3. Obliq uses, and supports, the Modula-3 thread, exception, and garbage-collection facilities. Its distributed-computation mechanism is implemented using Modula-3 Network Objects, allowing transparent support for multiple processes on heterogeneous machines. An Obliq computation may involve multiple threads of control within an address space (process), multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet.

The guiding principle that separates Obliq from other distributed procedural languages is its adherence to lexical scoping in a distributed higher-order context. This principle is conceptually simple and has a number of interesting consequences: it supports a natural and consistent semantics of distributed computation, and it enables elegant techniques for distributed programming. Lexical scoping ensures that the binding location of every identifier can be determined by simple analysis of the program text surrounding the identifier. Therefore, the meaning of program identifiers can be determined when they are introduced, not when they are used, allowing programmers to reason about the behavior of their programs, even when they are widely distributed and involve many simultaneous threads of control.

It does not matter where an identifier is used, since it always refers to the binding location *and* network site at which it was created. This is especially important when higher-order functions with free identifiers are transmitted over the network. Lexical scoping implies that these free identifiers are bound to variables when the higher-order function is analyzed, not when the function is executed. Therefore, higher-order functions are always self-contained as they move around the network, carrying along references to the variables referenced by their free identifiers.

Obliq supports uniform semantics across all data types, including objects, arrays and variables. As we noted during the discussion of the Shared Objects package in Section 3.5.2, and as Wilson and Bal point out in their evaluation of Orca, this ability to

share not only objects, but arrays and variables, simplifies many standard programming tasks. Unfortunately, unlike the Shared Objects package and Orca, Obliq does not support replicated data; all Obliq data values have client-server distribution semantics because they are built on top of the Network Objects package.

Repo is a descendant of Obliq that extends the Obliq data model to include both synchronized and unsynchronized replicated objects. Therefore, Repo objects have state that may be local to a site (as in Obliq) or replicated across multiple sites. The syntax and semantics of Repo differs as little as possible from Obliq, although the addition of replicated data does involve some conceptual differences. We will discuss the changes to the semantics of Obliq in Section 4.3, and to the syntax in Section 4.4. There are also a number of differences between Repo and Obliq that are unrelated to these enhancements to the type system, which will be discussed in Appendix D.

4.3 Distributed Semantics

As discussed above, Repo is a descendant of Obliq that extends the Obliq object model to include replicated objects, both synchronized and unsynchronized. In this section we will discuss the distributed semantics of Repo, focusing on how they differ from Obliq as a result of the addition of replicated data. In this discussion, a network address is a pair consisting of a *site address* (the process running on some machine) and a *memory address* at that site. The semantics of Obliq data can be described consistently by considering all addresses to be unique network addresses. Obliq data structures are assembled out of network addresses, just like ordinary data structures are assembled out of local addresses (more precisely, the implementation is designed to create this illusion). As data structures are passed around the network, the embedded network addresses do not change. For example, if an object is passed to another site, the value received at the remote site is a network address referring to the object at the original site. Data items can be explicitly copied between sites (creating new objects at new network addresses), but are never copied implicitly.

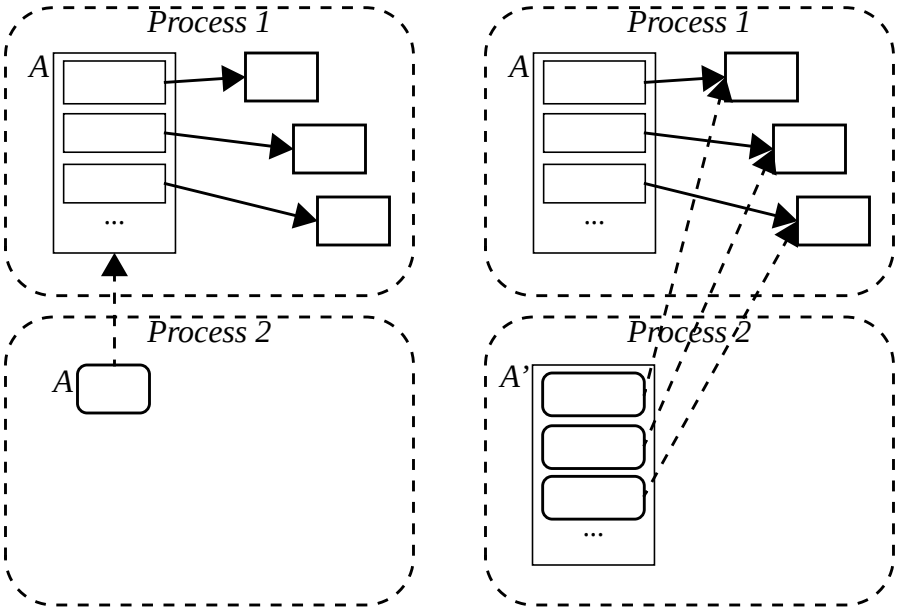
The semantics of Repo data are slightly more complicated because of the introduction of replicated data. Repo supports the following three distribution semantics when objects are transmitted from one site to another:

- *remote* objects, whose state exists at one site and are accessed remotely via remote method calls. In Obliq, all objects are remote.
- *replicated* objects, whose state is replicated at all sites that have references to them, with consistency enforced across all sites by ensuring all updates are applied in the same order to all replicas. When transmitted between sites, these objects are implicitly copied and new network addresses are created.
- *simple* objects, whose state is replicated at all sites to which they are transmitted, but do not have consistency enforced across these sites. When transmitted between sites, these objects are implicitly copied and new network addresses are created.

In Repo, we use the term *replicated* to refer to synchronized replicated objects, and the term *simple* to refer to unsynchronized replicated objects. We selected these terms because Obliq already used the term *synchronized* to refer to objects with an implicit mutex around all method calls. The term *simple* arises from the fact that these objects correspond to the simplest of all possible distribution semantics, in which data is copied between sites with no further action required.

As mentioned above, when Obliq data is transmitted around the network, the network addresses embedded in the data do not change, always referring to the original data item at the original site. Repo objects, however, can have embedded network references to replicated data. When a reference to an unsynchronized replicated data item is transmitted across the network, a new copy of the data referred to, with a new network address, is created at the destination site. Therefore, any embedded network references to this unsynchronized replicated data will be changed to refer to the new local address. If a reference to the same unsynchronized replicated data item is sent to a process multiple times, multiple new, independent replicas will be created.

When a reference to a synchronized replicated data item is transmitted across the network, the system first checks to see if a replica of this object exists in the destination



(a) Object *A* has client-server semantics

(b) Object *A* has replicated semantics

Figure 4-1: The effect of different distribution semantics. When an object *A* is copied from *Process 1* to *Process 2*, the result depends on the distribution semantics. For simplicity, assume all of the embedded references in object *A* are to client-server objects. In (a), *A* is a client-server object, so the network address is copied to *Process 2*, and all access to object *A* refers back to the original object. In (b), *A* is a replicated object, so a new replica is created and the embedded references in *A* are copied recursively. In this case, since the references are to client-server objects, their network addresses are copied and all access refers back to the original objects.

site. If a replica exists, its network address is substituted for any embedded references to this data object. If a replica does not exists, a new replica, with a new network address, is created and substituted for any embedded references to this data object. All replicas of a synchronized object maintain an association with each other, even though they have different network addresses.

Consider the following example, to help clarify the differences in the semantics, illustrated in Figure 4-1. Assume we have an array that we wish to distribute to a number of processes. If the array has client-server semantics, when references to it are passed around the network, only its network address is distributed, and all access is to the original array. If the array is replicated, when references to it are passed around the network, it is replicated. The process of replication causes its elements to be sent to the new site, which

causes the process to be repeated recursively: if an element is a client-server data value, only its network reference is sent to the new site, but if the element is a simple constant or a replicated data value, it is copied to the new site, with its elements in turn copied recursively, and so on. Since arrays and objects with different semantics can be mixed arbitrarily, interesting and powerful data structures can be built in a few lines of code.

The different distribution semantics also manifest themselves to the programmer by weakening the guarantee of correct execution that Obliq provides: in Obliq, computations are guaranteed to give the same result no matter where they are executed on the network[1]. Obliq can provide this guarantee because of the use of client-server data and lexical scoping: when program code is evaluated (either within object methods or procedures), its free variables are bound to data items and the network addresses of those data items are embedded in the function closure (the data structure representing the evaluated code). As the closure is passed around the network, it carries these network references with it, and they refer to the same data objects no matter where the closure is executed. Therefore, evaluating this closure always gives the same results, independent of the execution site.

The introduction of unsynchronized replicated data weakens this guarantee. If a function closure is sent to a remote site for execution, and some of its free variables are bound to unsynchronized data, those data values will be replicated at the remote site and the new network addresses substituted for the old ones in the closure. If the function does not modify the data object, the correct execution guarantee holds. However, if the function modifies one of these data items, the replicas at the original site will not reflect these changes, resulting in program execution that differs depending on the execution site (since data at different network addresses is being modified).

While programmers need to be careful when they use unsynchronized replicated data, the loss of this correctness guarantee is largely a pedagogical one; this guarantee is primarily a useful way of explaining and understanding how lexical scoping affects pro-

---

1. While this guarantee is useful, it is a simplistic one, since it necessarily assumes no built-in libraries are accessed that give different results at different sites. For example, if a computation accesses the file system, it may not find the same files at different sites.

objects:	{x_1 => a_1, ... ,x_n => a_n} every field of an object has state access: a.x, a.x(a_1, ... ,a_n) update: a.x := b, delegate a to b end
arrays:	[a_1, ... , a_n] every element of an array has state access: a[n] update: a[n] := b
variables:	var x = a variables have state (identifiers declared by "let" do not) access: x update: x := b

Table 4-1: Entities with state in Obliq. There are three kinds of entities that have state in Obliq: objects, arrays and variables. These entities are declared, accessed and updated as shown. The `delegate` update syntax redirects the fields of `a` to access the fields of object `b` (in this case), and is used to support a simple form of object migration.

gram behavior. We will return to this point in Section 4.8. The primary reason unsynchronized replicated data is provided is for efficient access to immutable data objects, which (by definition) will not be modified. We have also found other uses for unsynchronized data, some of which will be shown in Section 4.6.

4.4 Replication Syntax

When Repo was original designed, we made a decision to retain as much of the Obliq syntax as possible, with the goal of having all Obliq programs be valid Repo programs. With one small exception (that is unrelated to data replication, and will be discussed in Appendix D.1), we succeeded. In this section we will describe the differences between Obliq and Repo syntax introduced to support replicated data, including new declaration syntax, support for changing distribution semantics while cloning data, and new facilities for user-defined object picklers (analogous to those discussed in Section 3.4.1.3).

4.4.1 Declarations

Repo syntax differs from Obliq syntax primarily in the way data items are declared. In Obliq, there are three kinds of data items that can have state (shown in Table 4-1), and are

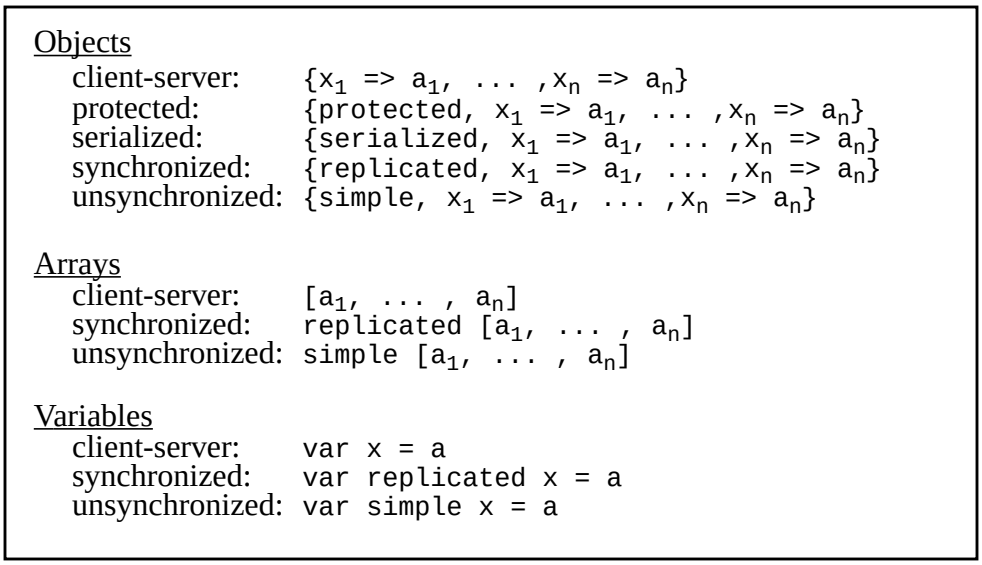


Table 4-2: Declaring entities with state in Repo. Repo has the same three kinds of entities with state as Obliq: objects, arrays and variables. These entities are accessed and updated in the same was as they are in Obliq. By default, these entities have Obliq's client-server distribution semantics. Additional keywords are used to declare synchronized and unsynchronized replication semantics, as shown. If an object is declared to be *protected*, its data fields can only be changed internally by its own methods and it cannot be cloned. If an object is declared *serialized*, there is an implicit lock around its methods that limits access to one thread at a time. Replicated objects can also be declared as protected and are automatically serialized (using the Shared Object mutex). Simple objects can be declared as protected and/or serialized.

thus affected by the addition of support for replication. These declarations are also valid Repo declarations, and create client-server entities.

To allow programmers to select different distribution semantics, we added the `simple` and `replicated` modifiers to these declarations, as shown in Table 4-2.[2] Since replicated objects are implemented using the Shared Objects package, we also need to decide which actions update these entities. In the case of arrays and variables, the decision is straightforward and intuitively obvious: the access operations shown in Table 4-1 read from the entities, and the update operations update them. In the case of objects, the decision is slightly more complex.

2. The exact positioning of the keywords in the declarations of Figure 4-2 was determined by the implementation of the Obliq interpreter.

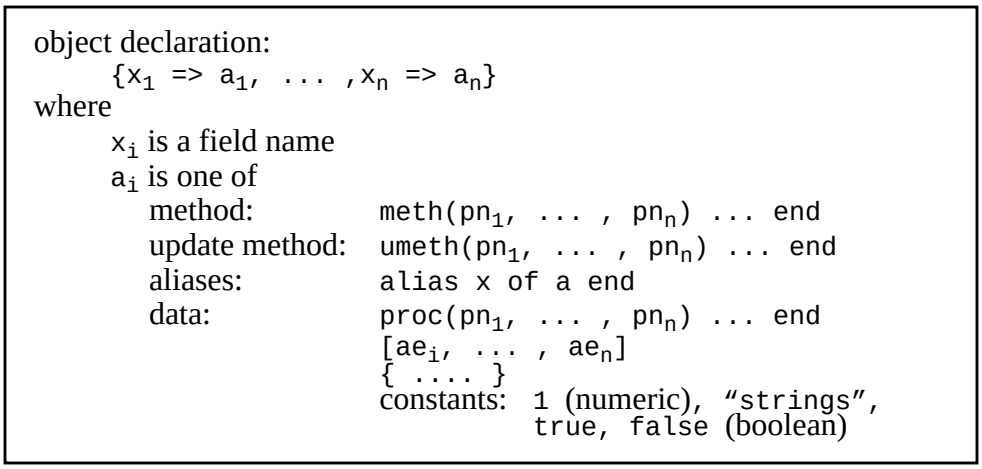


Figure 4-2: Declaring objects in Repo. Repo objects have four kinds of fields: methods (declared with the `meth` keyword), update methods (declared with the `umeth` keyword), aliases (that redirect the access to a field of a different object) and data values (all other fields). Data value fields can contain constants, arrays, objects or closures (higher-order functions declared with the `proc` keyword).

The access operations for objects shown in Table 4-1 correspond to reading fields and invoking methods, while the update operation corresponds to changing the value of a field. Like arrays and variables, we define the operations of reading and updating fields as read and update actions, respectively. Notice that this differs from the way Shared Objects are defined in Modula-3; since we define the action of updating a field as an update operation, if the object is synchronized, this update will be distributed to all replicas. Updates to the fields of Shared Objects are not normally distributed in this way. However, being able to create simple replicated objects without the need to define methods to update the data fields is convenient, and if the programmer wishes to restrict access to the data fields, they can declare the object as `protected`, which prevents the data fields from being modified from outside the object methods. Alternatively, lexical scoping can be used to define object data that is not contained in the object fields, and can therefore not be accessed from outside of the object.

The other access operation on an object is method invocation. As with Shared Objects at the Modula-3 level, methods are the primary means of updating and accessing objects. To differentiate between methods that update an object, and those that do not, we added an update method declaration, denoted with the `umeth` keyword. Methods created

with the original Obliq method syntax, denoted with the `meth` keyword, are treated as read methods, and those defined using the new `umeth` keyword are update methods, and are therefore applied to all replicas of the object.

4.4.2 Cloning Data

In Obliq, once an object is declared it cannot have fields added to it, nor can it be moved from the site at which it was created. However, Obliq supports object *cloning*. When an object is cloned, a new object is created with the same field names, and the fields are initialized to refer to the same values (methods, data or aliases) as the original object. Multiple objects can be cloned together to form a single new object, with the restrictions that all of the field names must be unique across the set of objects. Similarly, arrays cannot have their size changed. To change the size of an array, it must have a second array concatenated to it to create a new array containing the elements of both[3]. The new object or array is created at the site where the operation is executed, which need not be the same site as that of the objects or arrays being copied.

The decisions to have objects and arrays be immobile and structurally immutable were made to simplify the implementation and to keep the language clean and predictable. Cloning objects and concatenating arrays result in the creation of new data elements. If the old elements are in use, they will continue to exist unchanged; if they are not longer used, they will eventually be garbage collected.

In Repo, we must define what happens when multiple objects are cloned, or when multiple arrays are concatenated, and they do not all have the same distribution semantics. For example, what happens when we concatenate a remote array (`a1`) to a replicated array (`a2`) (i.e. `a3 := a2 @ a1`)? When concatenating arrays, we have the new array adopt the semantics of the array to which it is concatenated: in this example, the result (`a3`) is a replicated array. The decision is not so simple with objects, because we need a way of specifying update methods for replicated objects: for example, if we clone a simple object

3. A copy of an array that is the same size can be created by concatenating the array to the empty array. New arrays are also created by extracting a subarray of an array, but we will only refer to concatenation for simplicity.

to a replicated object to create a replicated object, we may want some of the fields of the simple object to be considered update methods in the resulting replicated object.

Therefore, we require that all objects have the same semantics if they are to be cloned together, and provide operators to convert an object from one distribution semantic to another. These new operations (`remote(a1)`, `replicated(a1,` *umeth-list*`)`, and `simple(a1)`) do not modify the semantics of an existing objects; rather, they each take their object argument (`a1`) and return a clone of that object with the appropriate distribution semantics (client-server, synchronized replicated or unsynchronized replicated, respectively). In addition, the `replicated` operator takes a second parameter, which is a list of the field names of methods to be converted from methods (created with the `meth` keyword) to update methods (that would have been specified with the `umeth` keyword had this object been originally created as a replicated object). For example, consider the following object:

```
let o1 = {simple,
          data => 1,
          get => meth (s) s.data end,
          set => meth (s, val) s.data := val end};
```

We could create a replicated version of this object as follows:

```
let o2 = replicated (o1, [“set”]);
```

This would give us the same object as this definition:

```
let o2 =
     {replicated,
      data => 1,
      get => meth (s) s.data end,
      set => umeth (s, val) s.data := val end};
```

For convenience, we also allow arrays to be used as arguments to these three conversion operators, in which case all three of the operators take the array to be cloned as their single parameter.

4.4.3 User-defined Picklers

In Obliq, copying an object from one site to another is always the result of an intentional action by the programmer (either cloning an object or concatenating an array). Therefore, it is left to the programmer to control what data is copied between processes when they create new objects or arrays in different processes.

In Repo, on the other hand, replicated data can be copied implicitly when data is passed between machines. Therefore, we need to provide some way for programmers to control what is copied, analogous to the Modula-3 custom pickling routines that were discussed in Section 3.4.1.3. To support pickling of replicated objects, we define the `objectpickler` command. In Modula-3, pickling is done by two routines, one that writes the object to a byte stream, and one that reads the object from a byte stream. Rather than write two routines for reading and writing objects from and to byte streams, which would be cumbersome and inefficient in Repo, the programmer creates two simple objects for reading and writing the object, with a field in these objects for each data field in the object being pickled. The syntax is as follows:

```
objectpickler objectreader read-actions-object
                    writer write-actions-object;
```

Read-actions and *write-actions* are the simple objects that define how an object is to be pickled. For each data field in *object*, these two objects must have a corresponding method that takes a single parameter and returns a Repo value. When the object is being pickled out to the network, the writer object methods are passed the current value of the corresponding field in the object, and the return value is written to the network. When the object is being pickled in from the network, the reader object methods are passed the value that was read in from the network (the value returned by the corresponding writer object method), and the return value is assigned to the corresponding field in the object. As an example, consider the custom pickler that is used by the replicated mutex in Figure 4-8:

```
objectpickler ret
   reader {simple,
           cv => meth (s,c) thread_condition() end,
           q => meth (s,q) q end}
   writer {simple,
           cv => meth (s,c) ok end,
           q => meth (s,q) q end};
```

This pickler is needed because the condition variable in the `cv` field cannot be passed across the network. Therefore, the `cv` method in the writer ignores its `c` parameter (the condition variable) and returns `ok` (which is the Repo null value). The `cv` method in the reader ignores its `c` parameter (the `ok` value read from the network) and creates a new condition variable and returns it. Thus, each replica of the object has its own local condition variable. The `q` field is to be passed across the network without being modified.

4.5 The Replication Module

In addition to the changes to the language syntax and semantics required by the addition of replicated data, we added a built-in `replica` module to Repo to give the programmer additional control over replicated objects. This module defines the Repo equivalents of the two Shared Object exceptions `SharedObj.Error` and `SharedObj.Fatal`, which

are `replica_failure` and `replica_fatal`, respectively. It also provides functions to create and destroy the Repo equivalents of the Shared Object callbacks, `replica_notify` and `replica_cancelNotifier`. The `replica_notify` function takes the replicated object and a simple (unsynchronized replicated) object to use as its notifier. The methods of the notifier object correspond to the pre and post updates that the programmer wishes to be informed of. See Section 4.6.2 for an example of this module in use. Finally, the module also exposes the Shared Object runtime routines to flush the update queue, as described in Section 3.4.2.1.

4.6 Examples

In this section we will give a number of examples to show the simplicity and flexibility of Repo's object distribution semantics. First, in Section 4.6.1 we will show how the simple tracker report distribution object from Section 3.4.1 can be reimplemented straightforwardly in Repo, and extended in a number of interesting ways in a small number of lines of code. Will we then use these simple objects to illustrate how a programmer would request notification of changes to the replicated objects in Section 4.6.2. Next, in Section 4.6.3 we take an example from one of our augmented reality prototypes that shows how flexible and general purpose data distribution allows simple distributed applications to be built with a minimum of fuss.

In Section 4.6.4 we will address the issue of distributed synchronization, showing how a simple distributed mutual exclusion lock (mutex) can be implemented using replicated objects. Finally, in Section 4.6.5 we will discuss the design of a hierarchical object directory that we have used as a central data structuring mechanism in some of our a distributed virtual environment prototypes.These examples illustrate how all three distribution semantics are important and can be linked together in straightforward and powerful ways.

4.6.1 Simple Tracker Report Distribution

In Section 3.4.1, we showed how to implement a simple replicated `TrackerPosition` object using the Shared Object package. The public interface for this object is shown in

```
let rep = {simple,
     x => 0.0, y => 0.0, ,z => 0.0};

let trackerDist = {replicated,
     data => rep,
     set => umeth (self, val)
          self.data := val;
        end,
     get => meth (self)
          self.val;
        end
   };
```

(a) Repo objects mimicking *TrackerPosition.i3*.

```
let rep = {simple,
     x => 0.0, y => 0.0, ,z => 0.0};

let trackerDist = {replicated,
     data => rep
  };
```

(b) An alternate implementation, without methods.

```
let rep = {simple,
     x => 0.0, y => 0.0, ,z => 0.0};

let trackerDist = {replicated,
     set => umeth (self, val) end
  };
```

(c) A stateless implementation.

Figure 4-3: An example of synchronized replicated objects in Repo. This simple example is based on the tracker distribution object in Figure 3-4. An object analogous to the one implemented in Modula-3 is shown in (a), with a prototype report object `rep`. Of course, any valid Repo object could be used here, but this object is shown for clarity. Notice that `rep` is a *simple* (unsynchronized replicated) object. Since data fields of Repo's synchronized replicated can be manipulated directly, the object could also be defined with no methods, as in (b). Finally, in (c) a stateless version of the object is shown, in which the data is distributed via the `set` method but is not stored in the object.

Figure 3-4 and the private interface and implementation are shown in Figure 3-5. In this section, we will show how this object, and a few variations of it, can be implemented with Repo. The version of the `TrackerPosition` object, called `trackerDist`, that corresponds most closely to the object in Chapter 3, is shown in Figure 4-3(a). The most obvious difference between the `trackerDist` and `TrackerPosition` is the amount of code required: `TrackerPosition`, the Modula-3 version, used almost 48 lines of code spread over three files, while `trackerDist`, the Repo version, required just 9. This

translates into an immediate times savings during implementation. While both objects are easy to understand, the clarity of the Repo implementation is significantly greater due to the terseness of the code. Of course, in Repo we lose the data hiding and separation of interface from implementation that exists in Modula-3, but for prototyping and exploratory programming, this is not a significant issue.

In Figure 4-3(b), we see an even simpler version of `trackerDist` that takes advantage of the fact that updates to the data fields of a synchronized replicated Repo object are distributed. In this case, since the only purpose of the two object methods was to read and write the `data` field, we can remove the methods and allow the programmer to simply access the field directly.

The final variation of the `trackerDist` object highlights an interesting feature of the replication model. Occasionally, when an object such as this is created, the programmer only cares about changes to the data field, but never accesses the data directly. In this case, we can create an object with a `set` method that does nothing with its argument. When this method is invoked, the arguments are marshalled and distributed to all processes containing replicas of the object, where the method is invoked on the replicas, causing any callback objects to have their appropriate methods invoked as well, as discussed in Section 4.6.2. In effect, one can view such objects as simple message ports: calling the `set` method sends a message to all replicas informing them of the new value of the data item, with the callback objects being used to receive these messages.

4.6.2 Asynchronous Change Notification

One of the features of the Shared Object package, and thus of Repo's synchronize replicated objects, is that they support asynchronous change notification. The change notification is analogous to that provided in Modula-3, and described in Section 3.3.3. An example of change notification is shown in Figure 4-4. Notice that the structure of the notifier objects is not substantially different than that of Modula-3 notifier objects. Change notification is supported by functions in the `replica` module, discussed in Section 4.5.

```
let trackerCB = {simple,
     pre‘set => meth (self, obj, val)
        (*do something before set is called*)
        true;
       end,
     post‘anyChange => meth (self, obj)
        (* do something after any update *)
       end
  };

replica_notify(trackerDist, trackerCB);
```

(a) Repo callback notifier for objects in Figure 4-3(a) and (c). This object is equivalent to the one in Figure 3-6 (b).

```
let trackerCB = {simple,
     post‘data => meth (self, obj, val)
        (*do something after data changes*)
       end
  };

replica_notify(trackerDist, trackerCB);
```

(b) Repo callback notifier for objects in Figure 4-3(b).

Figure 4-4: An example of notifier callback objects in Repo. Callback objects for the Repo tracker distribution objects in Figure 4-3. Notice that they are very similar in content to the Modula-3 callback objects shown in Figure 3-6 and discussed in Section 3.3.3. The only noticeable difference is that the names use a backquote (`‘`) instead of an underscore (_) after the `pre` and `post` keywords in the method names (e.g., `pre‘set` instead of `pre_set`). This is because underscore is not a valid character in identifiers, with backquote begin used where underscore normally would be.

4.6.3 Multi-person Spaceframe Construction

In this section we present a simple example of how Repo’s general purpose data sharing satisfies our goal of supporting exploratory programming of distributed augmented environments. The example illustrates both the ease of sharing arbitrary application state and the simplicity of modifying an application to share previously unshared state.

As part of the Augmented Reality for Construction (ARC) project, we built an AR system to assist with the construction of space frame buildings. Our system prompted the worker by displaying the next part to be installed in the correct location on the partially completed space frame, as shown in Figure 4-5(a). This prototype was built with the initial version of Coterie that did not have distributed object support, and is discussed in

(a) What the worker sees through their head-worn display during the installation of strut 11 (see Figure 2-8).

(b) What a remote consultant sees on their desktop display when the worker is installing strut 10.

Figure 4-5: Extending the space frame prototype for remote consultation. To experiment with the idea of having a remote expert consult with a worker, we implemented a simple remote viewer for our space frame construction assistant. In this version, the remote viewer can change the construction step the worker is performing.

Section 2.6. After this support was added, via the creation of Repo, we wished to explore how an AR construction assistant could be leveraged in other ways, ranging from monitoring the progress of the project to allowing workers to discuss problems with a remote expert. The first step in this exploration was to create a new visualization of the construction site, showing the status of the space frame, the location of the worker and the next piece they should install, as shown in Figure 4-5(b).

The new visualization prototype first needed to share the state of the construction task with the ARC prototype. Therefore, we modified the ARC prototype to move its single state variable (`step`, representing the current task step) into a replicated object, and exported this variable to the network. We imported this variable into our remote monitoring prototype, and allowed both programs to change the construction step.

However, we noticed that this did not give us all the information a remote monitor would need, especially information about when the worker performed incorrect actions. To distribute this information, we added routines to the replicated object that are called when various interesting conditions are noticed, shown in Figure 4-6(a). These conditions include the task being completed (`done`), the user scanning the wrong part

```
let initialStepNumber = 28;
....
let stepObj = {replicated,
       step => initialStepNumber,
       done => umeth (s) s.step := -1 end,
       wrongPart => umeth (s, barcode) end,
       wrongPosition => umeth (s) end
     };
```

(a) The replicated state variable. In our demonstration setup, the worker starts with the spaceframe partially completed, which is why the initial step number is 28.

```
let stepCB = {simple,
     post`step => meth(s,o,v)
          goToStep(v);
        end,
     post`done => meth(s,o)
          let oldGo = dummy.findName(stepName);
          if oldGo isnot ok then
             dummy.remove(oldGo);
             dummy.add(congratsText);
             congratsText.setName(stepName);
          end;
          goToStep(-1);
        end,
     post`wrongPart => meth(s,o,v)
          doWrongBC(v);
        end,
     post`wrongPosition => meth(s,o)
          doWrongPos();
        end
   };

let stepNotify = replica_notify(stepObj, stepCB);
```

(b) The notifer for the replicated state variable.

Figure 4-6: The replicated state for the distributed ARC prototype. By moving the state of the ARC prototype into a replicated variable, we can share it between multiple processes. Each process can create a notifier variable, similar to the one shown in (b) to perform whatever action is desired to react to the change. In this example, taken from the prototype code, the notification methods contain a mixture of inline code and calls to other procedures defined elsewhere in the code.

(`wrongPart`) and the user scanning the correct part in the wrong location (`wrongPosition`). Notice that the `wrongPart` and `wrongPosition` methods do nothing; they are simply used to distribute a message, which can be noticed and reacted to in a callback notification object (as is done in Figure 4-6(b)).

This example illustrates the simplicity of prototyping with Repo. Modifying the code to access the construction step variable from the replicated object was trivial, as was

adding the `wrongPart` and `wrongPosition` notification methods. In addition, none of this information is “typical” distributed virtual environment data, of the sort that would be supported by a distributed VE toolkit, but Repo allows us to distribute the information and react to changes in the various programs in a few lines of code that took a few minutes to write.

4.6.4 Distributed Mutexes

In the original design of the Shared Object package, we had intended to support locking objects globally. The main motivation for this feature was to allow the programmer to invoke multiple methods atomically by ensuring no update methods aside from their own would be executed while the object was locked. However, this facility was never implemented, partially because the implementation effort was non-trivial, but mostly because the need for it rarely arises: since locking an object globally is expensive, it is more efficient to implement a new method that encapsulates the functionality of a set of existing methods.

However, while this approach allows programmers to circumvent the need to atomically call multiple methods on a single object, it is not possible to use similar techniques to atomically invoke methods on a group of objects. While there are a number of ways this could be done, the simplest approach is to create a distributed mutual exclusion lock (mutex). In this section, we will describe a number of ways of creating a distributed mutex in Repo, and use this example to illustrate a number of useful programming techniques. This example is also important since the Shared Object replication model does not provide any distributed synchronization primitives, leaving it to the programmer to implement them. It is therefore instructive to see how this might be done.

A simple implementation of a distributed mutex is shown in Figure 4-7, and is implemented using a client-server object that contains a normal mutex. The object can be transmitted to any number of sites, and when they want to acquire the lock they will call back to the original object and attempt to acquire the local mutex. Similarly, to release the mutex, they will make a remote call and release the local mutex at the original site.

```
module mutex;
let new = proc ()
     let ret = {
        mu => thread_mutex(),
        acquire => meth (s)
             thread_acquire(s.mu);
          end,
        release => meth (s)
             thread_release(s.mu);
          end
     };
     ret;
   end;
end module;
```

Figure 4-7: A simple client-server mutex. The external routines (`acquire` and `release`) allow the mutex to be acquired and released by any site. The mutex exists at one site and all other sites make remote method calls to the object.

This implementation is fine for simple programs, but suppose we wanted to implement a mutex that represents a lock around some important piece of state, and we wanted to include information about the state of the mutex on a graphical display, perhaps showing if the lock is held and who holds its. In this case, it would be more appropriate to implement the mutex with a replicated object, so that the state is local and can be queried at any time, and callback notification objects can be used to asynchronously notify the user when the state of the mutex changes. A simple implementation using a replicated object is shown in Figure 4-8. This object implements a mutex by having the acquire method enqueue an *id* in a replicated queue (the `q` data field) and waiting until its *id* has reached the head of the queue before returning to the caller. The release method simply removes the top element from the queue. We enqueue events in the distributed queue `q` to ensure the mutex is fair and will not give preferential access to sites close to the sequencer. An unfair implementation might have an internal update method that tries to acquire the mutex, and raises an exception if the mutex is held. The acquire method would wait until the mutex is released and then try again to acquire it. Such an implementation would give preferential access to sites that have a faster network connection to the sequencer for the mutex object, which is undesirable.

Notice that in both these implementations, the mutex can be released by any process, not just the one that acquired it. This may seem odd, but it is analogous to the behav-

```
module mutex;
let held = exception("held mutex");
let new = proc ()
     let ret = {replicated,
       cv => thread_condition(),
       id => meth (s)
            sys_address & "." & fmt_int(process_myId) &
                 "." & fmt_int(thread_id(thread_self()))
          end,
       q => [],
       enqueueId => umeth (s,id)
            s.q := s.q @ [id];
          end,
       dequeueId => umeth (s)
            s.q := s.q[1 for (#(s.q)-1)];
            thread_signal(s.cv);
          end

       acquire => meth (s)
            let id = s.id();
            if #(s.q) > 0 then
              if s.q[0] is id then raise(held) end;
            end;
            s.enqueueId(id);
            watch s.cv until (s.q[0] is id) end;
          end,
       release => meth (s)
            s.dequeueId();
          end
     };
     objectpickler ret
          reader {simple,
                  cv => meth (s,c) thread_condition() end,
                  q => meth (s,q) q end}
          writer {simple,
                  cv => meth (s,c) ok end,
                  q => meth (s,q) q end};
     ret;
   end;
end module;
```

Figure 4-8: A simple replicated mutex. The external routines (`acquire` and `release`) allow the mutex to be acquired and released by any site.

ior of a non-distributed mutex, which can be released by any thread, not just the one that acquired the mutex. In fact, the implementation in Section 4-7 would not work if the mutex had to be released by the same thread that acquired it, because there is no way to guarantee that the thread servicing the remote method call for the acquire would also service the remote method call for the release.

However, this feature creates a potential problem with both of these distributed mutexes: if the process holding the mutex terminates without releasing it, it will never be

unlocked. While this behavior is also analogous to a non-distributed mutex, it may be considered undesirable in many cases because processes in a distributed system are much more likely to terminate unexpectedly than threads in a process. Therefore, we have implemented a version of the replicated mutex in Appendix E that will release the mutex if the process that acquired the mutex terminates without releasing it. This version uses a client-server object (rather than a text string) as the `id` when acquiring the mutex, and if the `id` object becomes unreachable, the mutex is automatically unlocked.

The version of the mutex in Appendix E also implements another useful method, `tryAcquire`. This method attempts to lock the mutex, but if the mutex is locked, the method fails and raises an exception instead of blocking. The `tryAcquire` method is useful when creating distributed, interactive applications where you do not want to block the user-interaction for any significant length of time. For example, imagine implementing an interactive, multi-user graphical editor. If we want to ensure that only one person can manipulate an object at a time, we may use a distributed mutex such as this for each object. When the user clicks on an object to modify it, the system could issue a `tryAcquire` on that object's mutex. If the mutex is available, it would be locked, and the program could change the object to indicate that user may modify it. If the mutex is already locked, the `tryAcquire` would fail, and the program could beep (or issue some other notification), indicating that the object is not available for updating by the user.

4.6.5 Hierarchical Object Directories

In this section, we will describe how hierarchical *object directories* (HOD) would be implemented in Repo. The goal of the object directories is to provide a mechanism for structuring applications that is useful for both stand-alone and distributed applications. The HOD is similar in flavor to directories in a file system, the hierarchical environments used in dVS [Grimsdale, 1991], or the name spaces used in many different application domains. We will not include the code for our HOD, as it is fairly long.

The purpose of a HOD is to handle object lookup and management. It was designed to be analogous to a file-system, with a single object directory (OD) containing a set of key-value pairs that associate objects with textual names. An OD can contain refer-

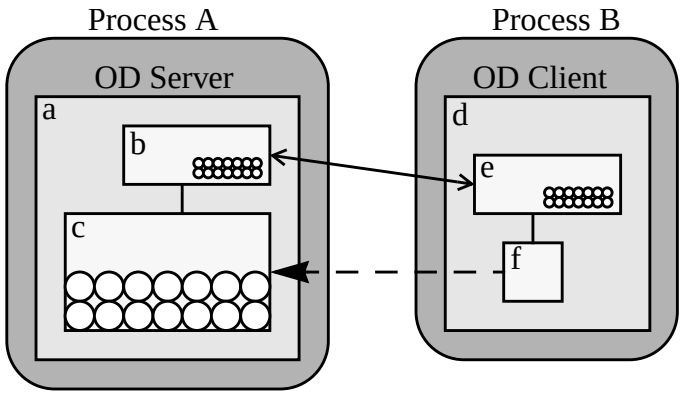


Figure 4-9: A single Object Directory (OD). The server on the left consists of (a) an unsynchronized replicated object acting as a wrapper, (b) a synchronized, replicated object implementing the notifier directory and (c) a client-server object implementing the storage directory. The client on the right has (d) its own copy of the wrapper object, (e) a shared copy of the notifier directory, and (f) a remote reference to the storage directory.

ences to other ODs, allowing arbitrary hierarchies to be created. References to virtually any kind of object can be stored in an OD. The OD can contain the actual objects or references to entries in other ODs (the equivalent to symbolic links in a file-system).

In addition to providing a general solution to the *naming problem* (how to meaningfully assign names to services and resolve those names to computer addresses) [Coulouris et al., 1994], the HOD can serve as the primary structuring metaphor for a family of distributed applications. By allowing an OD to contain references to other ODs, we can organize the HODs into a single global name space that allows applications to communicate with each other in a meaningful way. Within this hierarchy, data can be organized in well-defined subhierarchies so that applications know where to look for particular kinds of data and services. Furthermore, by allowing clients to watch one or more ODs for changes, such as the addition or deletion of entries, clients can react to changes in the world without the need for direct communication with the instigator of those changes. Such shared data-space techniques are widely used in AI blackboard systems and programming languages such as Linda [Carriero and Gelernter, 1992].

To build a simple OD, each of the three types of objects is used, as shown in Figure 4-9. The OD itself is implemented using an unsynchronized replicated *wrapper*

object (Figure 4-9(a) and (d)). This object has data fields and methods to implement the OD functionality. For example, there would be methods to add elements to, or delete elements from, the OD. In addition to any other incidental data, the OD contains references to two important objects in its data fields, a *storage directory* and a *notifier directory.* The *storage directory* is a client-server object that implements a centralized object store using key-value pairs (Figure 4-9(c)). It contains the actual data objects stored in the OD. The *notifier directory* is a synchronized replicated object that contains a small, constant-size piece of information for each entry in the directory, such as the type of the object, and is also used to receive notification of changes to the directory (Figure 4-9(b) and (e)).

Because the storage directory is implemented with a client-server object, it is not replicated. The single copy is accessed via remote method calls from any process that receives a copy of the OD. Conversely, since the notifier directory is implemented with a synchronized replicated object, it is fully replicated in all processes that receive a copy of the OD, with any updates to the OD distributed to it. There are three things to understand about why the OD is designed this way: what happens when an OD is passed to a remote process, how OD methods are implemented, and how Callback Objects are used by the OD.

First, consider what happens when an OD is passed from one process to another, as a parameter or return value of a client-server or replicated object method call. Since the OD is an unsynchronized replicated object, a new, independent copy of the object is created in the second process. As part of the process of creating that copy, the data fields of the object are copied using their semantics. Therefore, the new process will contain a new replica of the notifier directory, and a remote reference to the storage directory (Figure 4-9(f)). All of this happens automatically when a reference to the OD is transmitted to a new process.

Given the structure shown in Figure 4-9, how are OD methods implemented? Consider adding an element to an OD with a simple *put* method, "*put(name, object)*," which stores *object* in the OD under the key *name*. The *put* method of the OD would perform the following actions:

- store the object in the storage directory using the corresponding storage directory *put* method, and
- store the type of the object in the notifier directory using the corresponding notifier directory *put* method, which has been designated as a shared update method.

No matter which process performs the *put* operation, the outcome is the same:

- a remote procedure call is performed to store the object in the central storage directory, and
- the type of the object is stored in the shared notifier directory in all replicas, causing any Callback Objects registered for these replicas to have their *put* notifier methods invoked.

The wrapper object would use the Callback Objects associated with the notifier directory to monitor an object directory for changes on behalf of local clients that have requested notification when the OD changes.

4.7 Implementation

In this section, we will discuss the implementation of Repo. In particular, we will use the implementation of Repo as an example of using the Shared Object package to add replicated data to a complex application. Another interesting aspect of our experience implementing Repo, how we dealt with an efficiency problem of Obliq that was exacerbated by the addition of replicated data, will be discussed in Appendix D.3.

To create Repo, there were three kinds of objects (representing objects, arrays and variables) in the Obliq runtime that needed to be updated to support the two additional replicated distribution semantics. We shall look at the implementation of the Obliq and Repo array objects in detail, and then comment on an interesting aspect of the object representing Repo objects.

The definitions for the Obliq array object are shown in Figure 4-10. All objects that represent Obliq values inherit from the `Val` object. The array object `ValArray` is an Obliq value that contains a single field, `remote`, that holds a reference to a client-server

```
Val = BRANDED "Val" OBJECT END;
ValArray = Val BRANDED "ValArray" OBJECT
     remote: RemArray;
  END;

RemArrayServer <: RemArray;
RemArray = NetObj.T BRANDED "RemArray" OBJECT METHODS
     Size(): INTEGER RAISES {NetObj.Error};
     Get(i: INTEGER): Val RAISES {ServerError, NetObj.Error};
     Set(i: INTEGER; val: Val)
          RAISES {ServerError, NetObj.Error};
     Sub(start,size: INTEGER): ValArray
          RAISES {ServerError, NetObj.Error};
          (* Extract the subarray self[start for size]. *)
     Upd(start, size: INTEGER; READONLY other: REF Vals)
          RAISES {ServerError, NetObj.Error};
          (* Update self[start for size] with other[0 for size]. *)
     Obtain(): REF Vals RAISES {NetObj.Error};
          (* Return self.array if local, or a copy of it if remote.
             Modifying the result of Obtain may violate network
             transparency. *)
  END;
```

Figure 4-10: The internal definition of an Obliq array. `ValArray` is an Obliq value object, which contains a reference to the client-server object `RemArray`. `RemArrayServer` is the concrete local implementation of the array object that serves the remote method calls. Methods are invoked through the `remote` field.

Network Object, `RemArray`, implementing the actual array. When an instance of `ValArray` is transmitted over the network, the Network Object runtime system substitutes a proxy object for `remote` that redirects all access across the network to the original object. Obliq objects and variables are implemented in exactly the same way.

To convert the Obliq runtime to support replicated data, we added a level of indirection, as shown in Figure 4-11. In the Obliq runtime, when an array method (such as `Size` or `Obtain`) needed to be invoked, the code would explicitly dereference the `remote` field of the `ValArray` object. In the Repo runtime, the methods have been pushed out to the `ValArray` object, and the runtime code modified to invoke those methods directly without referring to the `remote` field, which is removed from the object. Three subtypes of `ValArray` are defined, which represent the three object semantics. Each of these objects has a field, analogous to the `remote` field of `ValArray` in the Obliq objects, that contains an object of the appropriate type (a Network Object for the client-server array, a Shared Object for the synchronized replicated array, and a plain

```
Val = BRANDED "Val" OBJECT END;
ValArray = Val BRANDED "ObValue.ValArray" OBJECT METHODS
     Size(): INTEGER RAISES {SharedObj.Error, NetObj.Error,
                             Thread.Alerted};
     Get(i: INTEGER): Val RAISES {SharedObj.Error, ServerError,
                                  NetObj.Error, Thread.Alerted};
     Set(i: INTEGER; val: Val) RAISES {SharedObj.Error,
                    ServerError, NetObj.Error, Thread.Alerted};
     Sub(start,size: INTEGER): ValArray RAISES {SharedObj.Error,
                    ServerError, NetObj.Error, Thread.Alerted};
     Upd(start, size: INTEGER; READONLY other: REF Vals)
          RAISES {SharedObj.Error, ServerError, NetObj.Error,
                  Thread.Alerted};
     Obtain(): REF Vals RAISES {SharedObj.Error, NetObj.Error,
                                Thread.Alerted};
   END;

ValRemArray <: ValRemArrayPublic;
ValRemArrayPublic = ValArray OBJECT remote: RemArray END;

ValReplArray <: ValReplArrayPublic;
ValReplArrayPublic = ValArray OBJECT replica: ReplArray END;

ValSimpleArray <: ValSimpleArrayPublic;
ValSimpleArrayPublic = ValArray OBJECT simple: SimpleArray END;

... RemArray is unchanged ...
... SimpleArray is defined analogously ...

ReplArrayStd <: ReplArray;
<* SHARED UPDATE METHODS ReplArrayStd.init, ReplArrayStd.Set,
    ReplArrayStd.Upd *>
ReplArray <: ReplArrayPublic;
ReplArrayPublic = SharedObj.T BRANDED
     "ObValue.ReplArrayServerPublic" OBJECT METHODS
     init (): ReplArray RAISES {SharedObj.Error};
     Size(): INTEGER RAISES {SharedObj.Error};
     Get(i: INTEGER): Val RAISES {ServerError, SharedObj.Error};
     Set(i: INTEGER; val: Val)
          RAISES {ServerError, SharedObj.Error};
     Sub(start,size: INTEGER): ValArray
          RAISES {ServerError, SharedObj.Error};
     Upd(start, size: INTEGER; READONLY other: REF Vals)
          RAISES {ServerError, SharedObj.Error};
     Obtain(): REF Vals RAISES {SharedObj.Error};
END;
```

Figure 4-11: The internal definition of a Repo array. We move the methods for accessing the array to the Repo value object `ValArray`, and create three subtypes for the three distribution semantics. The methods of these subtypes invoke the methods of their appropriate internal object (`remote`, `replica` or `simple`). The Obliq runtime code was modified to invoke the methods of the array value object (`ValArray`), instead of dereferencing the `remote` field directly. The code to generate new arrays also had to be changed to generate the appropriate subtype, but aside from these well defined changes, the code was not substantially modified.

```
ReplObjStd <: ReplObj;
<* SHARED UPDATE METHODS ReplObjStd.init, ReplObjStd.InvokeUpdate,
       ReplObjStd.Update, ReplObjStd.RedirectFields *>
ReplObj <: ReplObjPublic;
ReplObjPublic = SharedObj.T BRANDED "ObValue.ReplObjServerPublic"
  OBJECT METHODS
    init (): ReplObj RAISES {SharedObj.Error};
    Who(VAR(*out*) protected: BOOLEAN): TEXT
         RAISES {SharedObj.Error};
    Select(swr: SynWr.T; label: TEXT; VAR hint: INTEGER): Val
         RAISES {Error, Exception, ServerError, SharedObj.Error};
    Invoke(swr: SynWr.T; label: TEXT; argNo: INTEGER;
           READONLY args: Vals; VAR hint: INTEGER): Val
         RAISES {Error, Exception, ServerError, SharedObj.Error};
    Update(label: TEXT; val: Val; internal: BOOLEAN;
           VAR hint: INTEGER)
         RAISES {ServerError, SharedObj.Error};
    Redirect(val: Val; internal: BOOLEAN)
         RAISES {ServerError, SharedObj.Error};
    Has(label: TEXT; VAR hint: INTEGER): BOOLEAN
         RAISES {SharedObj.Error};
    Obtain(internal: BOOLEAN): REF ObjFields
         RAISES {ServerError, SharedObj.Error};
    ... other methods, needed by the reflection package ...
END;
```

(a) The replicated object definition.

```
REVEAL
  ReplObj = ReplObjPublic BRANDED "ObValueRep.ReplObjServerRep"
  OBJECT
    ... data fields ...
  METHODS
    InvokeUpdate(swr: SynWr.T; label: TEXT; argNo: INTEGER;
                 READONLY args: Vals; VAR hint: INTEGER): Val
      RAISES {Error, Exception, ServerError, SharedObj.Error} :=
           ReplObjInvokeUpdate;
    ... other internal methods ...
  OVERRIDES
    ... external method overrides ...
  END;
```

(b) Excerpts from the private part of the replicated object definition.

Figure 4-12: The internal definition of a Repo replicated object. This object is interesting because, unlike the array object in Figure 4-11, it makes use of internal update methods.

Modula-3 object for the unsynchronized replicated array). When these objects are transmitted across the network, the Network or Shared Object runtimes handle implementing the semantics.

Repo objects and variables are implemented in an analogous way. However, there is one detail of the implementation of Repo objects that is interesting, concerning the

implementation of update methods to Repo objects. If we look at the Modula-3 object representing Repo objects, shown in Figure 4-12, we see that there is one method for invoking Repo methods (`Invoke`). Since some Repo object methods need to be invoked as Shared Object non-update methods and some need to be invoked as Shared Object update methods, we add a new method (`InvokeUpdate`) for invoking update methods, and denote it as a Shared Object update method. However, since we do not want to require the runtime code to know if the method they are calling is an update method or not, we make the `InvokeUpdate` method an internal method, and add a check inside the `Invoke` method to see if the method being invoked is an update method. If it is, the `InvokeUpdate` method is called to handle the method invocation. In this way, the external interface of the `RepoObj` object is unchanged from Obliq to Repo, reducing the changes required in the rest of the code.

4.8 Usability of Repo

We have used Repo to build a number of prototypes, including those described throughout this dissertation, and our experiences have been mostly positive. The ability to quickly and effortlessly create distributed applications using arbitrary combinations of objects, arrays and variables with both client-server and replicated distribution semantics has allowed us to concentrate on the applications and the interaction techniques we are interested in exploring, and take the distribution of data largely for granted.

While programmers can also build data structures in Modula-3 that combine these distribution semantics, courtesy of the Shared and Network Objects packages, exploratory programming in an interpreted language such as Repo is significantly faster. Furthermore, Repo's dynamic type system, and the ability to distribute arrays and variables in addition to objects, gives the programmer greater flexibility. While Shared and Network Objects can be used just as normal objects, Modula-3's strong static typing combined with the requirement that these objects inherit from different distinguished types means that a programmer can not mix them quite so freely as objects can be mixed in Repo. For example, in Modula-3, a procedure must be defined to take one of a Network or Shared Object as a

parameter, but in Repo any data value can be passed as a parameter to the same procedure (it is up to the programmer to ensure that correct values are used in the correct locations).

However, there is a price to be paid for the increased flexibility of a dynamically typed language, and that is the greater difficulty in tracking down bugs; since procedures are untyped, incorrect usage may not cause errors immediately, since the variables themselves may not be used immediately. While the prototypes we have been creating have typically only been a few hundred to a few thousand lines of code, we have experienced problems debugging some of the larger ones. It was these problems that motivated us to create the reflection module (see Appendix D.2) to allow programmers to add type checking and controlled object access to their programs when they see fit. By judiciously checking parameters in a few key locations in a program, debugging of programs has been greatly simplified.

We have also learned some lessons about our design of Repo. One relates to the usefulness of the custom pickling facilities. It turns out that programmers need to be told that the pickling facilities in Repo are much less efficient than in Modula-3, and should not be used to try and obtain small performance improvements. In Modula-3, picklers are associated with objects of a certain type, and are compiled into all instances of the program. Therefore, aside from sending a small value to identify the type, only the data generated by the pickling routine is sent across the network. In Repo, on the other hand, there are no object types, so these pickling objects are associated with *instances* of Repo objects. Therefore, before these pickling object methods are run, the infrastructure must copy the basic object structure, including the pickling objects themselves and all the object methods, between processes. As a result, attaching custom pickling objects to an object initially increases the amount of information that is sent over the network, and is therefore useful primarily for situations, such as the example in Section 4.4.3, where correctness (i.e., a condition variable can not be copied over the network), rather than efficiency, is the motivation for creating the custom pickler.

Finally, we have noticed a recurring problem with novice programmers that has led us to desire changing our decision to aim for backward compatibility with Obliq. We have found that novice programmers tend to forget to specify the distribution semantics of data

values as they are programming, implicitly assuming that the default is unsynchronized replication because that is what they are used to in traditional programming languages. We suspect that if we required all data values (arrays, objects and variables) to have their distribution semantics specified, instead of having objects default to client-server sharing with replication as an option, novices would learn to think about the semantics of the objects they are creating more quickly, and experienced programmers would not introduce bugs into their programs by forgetting to specify the semantics. However, this problem, and the others mentioned in this section, are relatively minor, especially in relation to Repo's advantages for building distributed applications.

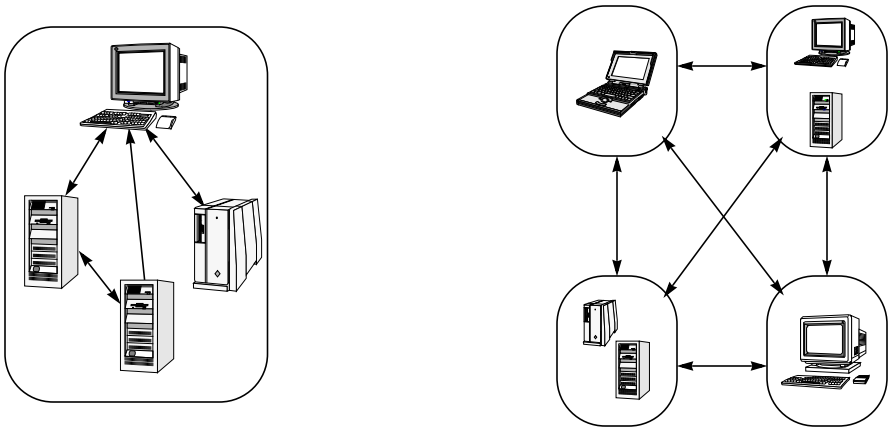
CHAPTER 5 Repo-3D

"It's part of the lattice of coincidence that lays on top of everything"

– Miller, from *Repo Man*

In the previous chapters, we have discussed the design of various components of Coterie. We started with the Shared Object package, a flexible system for exploratory programming of distributed interactive applications, and showed examples of its usefulness. We then discussed Repo, the interpreted language built on top of the Shared Object package, in which all Coterie applications are written. Repo presents the programmer with a distributed language and a set of libraries for doing various common tasks, such as interacting with trackers or file systems, or building 3D graphical scenes using Obliq-3D. (Obliq-3D was introduced in Section 2.5 and will be discussed in greater depth in Section 5.2.) In this chapter, we will discuss Repo-3D, a novel distributed graphics package that is the final significant component of Coterie.

Looking back at the prototypes described in Section 2.6, it turns out that the bulk of our development time, and the bulk of the resulting code, involved using Obliq-3D to create 3D graphical displays. Unfortunately, since Obliq-3D data structures are not directly distributed, if we use it to build distributed prototypes programmers will be forced to build their own distributed graphical data structures in Repo and synchronize them to the Obliq-3D scenes in each process. This is a tedious and error prone endeavor, and is contrary to our original goal of having distributed prototypes be as simple and straightforward to implement as non-distributed ones. To address this problem, we created Repo-3D, a successor to Obliq-3D in which most of the objects in the graphical scene are built using Shared Objects and are therefore directly distributable. Repo-3D is aimed at simplifying the creation of the graphical components of our distributed applications.



(a) (b)

Figure 5-1: Two meanings of *distributed graphics*. (a) a single logical graphics system with distributed components, and (b) multiple distributed logical graphics systems. We use the second definition here.

Traditionally, *distributed graphics* has referred to the architecture of a single graphical application whose components are distributed over multiple machines (e.g., our initial single-user AR prototypes, discussed in Section 2.1, as well as the work of others, such as [Fairen and Vinacua, 1997], [Holbrook et al., 1995], and [Phillips et al., 1989]) (Figure 5-1a). By taking advantage of the combined power of multiple machines, and the particular features of individual machines, otherwise impractical applications became feasible. However, as machines have grown more powerful, application domains such as Computer Supported Cooperative Work (CSCW) and Distributed Virtual Environments (DVEs) have been making the transition from research labs to commercial products. In addition, it is finally becoming feasible to experiment with more heavily distributed application domains, such as augmented environments (AEs). As a result, the term *distributed graphics* is increasingly used to refer to systems for distributing the shared graphical state of multi-display/multi-person, distributed, interactive applications (Figure 5-1b). This is the definition that interests us, and that we use here.

While many excellent, high-level programming libraries are available for building stand-alone 3D applications (e.g., Obliq-3D, Inventor [Strauss and Carey, 1992], Performer [Rohlf and Helman, 1994] and Java 3D [Sowizral et al., 1998]), there are no similarly powerful and general libraries for building distributed 3D graphics applications. All

CSCW and DVE systems with which we are familiar (discussed in Section 2.4) use the approach mentioned above: a mechanism is provided for distributing application state (either a custom solution or one based on a general-purpose distributed programming environment, such as ISIS [Birman, 1993] or Repo), and the state of the graphical display is maintained separately in the local graphics library. As we have found, keeping these "dual databases" synchronized is a complex, tedious, and error-prone endeavor.

This problem is similar to the "dual database" problem encountered when building non-distributed graphical applications, where application and graphical state must be maintained separately and manually synchronized by the programmer. The dual database problem is addressed by the designers of modern non-distributed 3D graphics libraries, such as Obliq-3D and Inventor, by allowing programmers to extend the graphical scene objects to encode application state. Extending this approach to a distributed context is the basis for our design of Repo-3D; Repo-3D's distributed objects can be extended to include application state, helping the programmer avoid the dual database problem when building distributed graphical applications.

However, no matter how simple the construction of a distributed application may be, a number of differences between distributed and monolithic graphical applications must be addressed. These include:

- *Distributed control*. In a monolithic application, a single component can oversee the application and coordinate activities among the separate components by notifying them of changes to the application state. This is not possible in a non-trivial distributed application. Therefore, we must provide mechanisms for different components to be notified of changes to the distributed state.
- *Interactivity.* Updates to distributed state will be slower than updates to local state, and the amount of data that can be distributed is limited by network bandwidth. If we do not want to sacrifice interactive speed, we must be able to perform some operations locally. For example, an object could be dragged locally with the mouse, with only a subset of the changes applied to the replicated state.

- *Local variations.* There are times when a shared graphical scene may need to be modified locally. For example, a programmer may want to highlight the object under one user's mouse pointer without affecting the scene graph viewed by other users.

Repo-3D addresses these problems in two ways. First, as with any Shared Object, a programmer can associate a Shared Object Callback Object with most Repo-3D objects. When combined with Repo's general purpose programming facilities, this allows reactive programs to be built in a straightforward manner. To deal with the second and third problems, we introduce the notion of *local variations* to graphical objects. That is, we allow the properties of a graphical object to be modified locally, and parts of the scene graph to be locally added, removed, or replaced.

In Section 5.1 we will discuss related work, followed by a more in-depth overview of Obliq-3D in Section 5.2. The design of Repo-3D is covered in Section 5.3, followed by some examples of Repo-3D in use in Section 5.4. The implementation will be discussed in Section 5.5, and some performance issues in Section 5.6. We will close the chapter with a discussion of our experiences using Repo-3D.

5.1 Related Work

There has been a significant amount of work that falls under the first, older definition of distributed graphics. A large number of systems, ranging from established commercial products (e.g., IBM Visualization Data Explorer [Lucas et al., 1992, IBM Corporation, 1993]) to research systems (e.g., PARADISE [Holbrook et al., 1995] and ATLAS [Fairen and Vinacua, 1997]), have been created to distribute interactive graphical applications over a set of machines. However, the goal of these systems is to facilitate sharing of application data between processes, with one process doing the rendering. While some of these systems can be used to display graphics on more than one display, they were not designed to support high-level sharing of graphical scenes.

Most high-level graphics libraries, such as UGA [Zeleznik et al., 1991], Inventor [Strauss and Carey, 1992] and Java 3D [Sowizral et al., 1998], do not provide any support for distribution. Others, such as Performer [Rohlf and Helman, 1994], provide support for

distributing components of the 3D graphics rendering system across multiple processors, but do not support distribution across multiple machines. One notable exception is TBAG [Elliott et al., 1994], a high-level constraint-based, declarative 3D graphics framework. Scenes in TBAG are defined using constrained relationships between time-varying functions. TBAG allows a set of processes to share a single, replicated constraint graph. When any process asserts or retracts a constraint, it is asserted or retracted in all processes. However, this means that all processes share the same scene, and that the system's scalability is limited because all processes have a copy of (and must evaluate) all constraints, whether or not they are interested in them. There is also no support for local variations of the scene in different processes.

Machiraju [Machiraju, 1997] investigated an approach similar in flavor to ours, but it was not aimed at the same fine-grained level of interactivity and was ultimately limited by the constraints of the implementation platform (CORBA and C++). For example, CORBA objects are heavyweight and do not support replication, so much of their effort was spent developing techniques to support object migration and "fine-grained" object sharing. However, their fine-grained objects are coarser than ours, and, more importantly, they do not support the kind of lightweight, transparent replication we desire. A programmer must explicitly choose whether to replicate, move, or copy an object between processes when the action is to occur (as opposed to at object creation time). Replicated objects are independent new copies that can be modified and used to replace the original—simultaneous editing of objects, or real-time distribution of changes as they are made is not supported.

Of greater significance is the growing interest of this sort of system in the Java and VRML communities. Java, like Modula-3, is much more suitable as an implementation language than C or C++ because of its cross-platform compatibility and support for threads and garbage collection: Without the latter two language features, implementing complex, large-scale distributed applications is extremely difficult. Most of the current effort in these communities has been focused on using Java as a mechanism to facilitate multi-user VRML worlds (e.g., Open Communities [Open Communities, 1997]). Unfortunately, these efforts concentrate on the particulars of implementing shared virtual

environments and fall short of providing a general-purpose shared graphics library. For example, the Open Communities work is being done on top of SPLINE [Waters et al., 1997], which supports only a single top-level world in the local scene database.

5.2 Obliq-3D: An Overview

Obliq-3D is composed of Anim-3D, a 3D animation package written in Modula-3, and a set of wrappers that expose Anim-3D to the Obliq programming language (see Section 2.5). Anim-3D is based on three simple and powerful concepts: *graphical objects* for building graphical scenes, *properties* for specifying the behavior of the graphical objects, and input event *callbacks* to support interactive behavior (these callbacks are unrelated to the Shared Object Callback Objects). Anim-3D uses the *damage-repair* model: whenever a graphical object or property changes (is damaged), the image is repaired without programmer intervention.

Graphical objects (GOs) represent all the logical entities in the graphical scene: geometry (e.g., lines, polygons, spheres, polygon sets, and text), lights and cameras of various sorts, and groups of other GOs. One special type of group, the `RootGO`, represents a window into which graphics are rendered. GOs can be grouped together in any valid directed acyclic graph (DAG). The GO class hierarchy is shown in Figure 5-2.

A *property* is defined by a *name* and a *value*. The name determines which attribute is affected by the property, such as “Texture Mode” or “Box Corner1”. The value specifies how it is affected and is determined by its *behavior*, a time-varying function that takes the current animation time and returns a value. Properties, property values, and behaviors are all objects, and their relationships are shown in Figure 5-3. When a property is created, its name and value are fixed. However, values are mutable and their behavior may be changed at any time. There are four kinds of behaviors for each type of property: *constant* (do not vary over time), *synchronous* (follow a programmed set of *requests*, such as “move from A to B, starting at time t=1 and taking 2 seconds”), *asynchronous* (execute an arbitrary time-dependent function to compute the value) and *dependent* (asynchronous properties that depend on other properties). Synchronous properties are linked to *animation handles* and

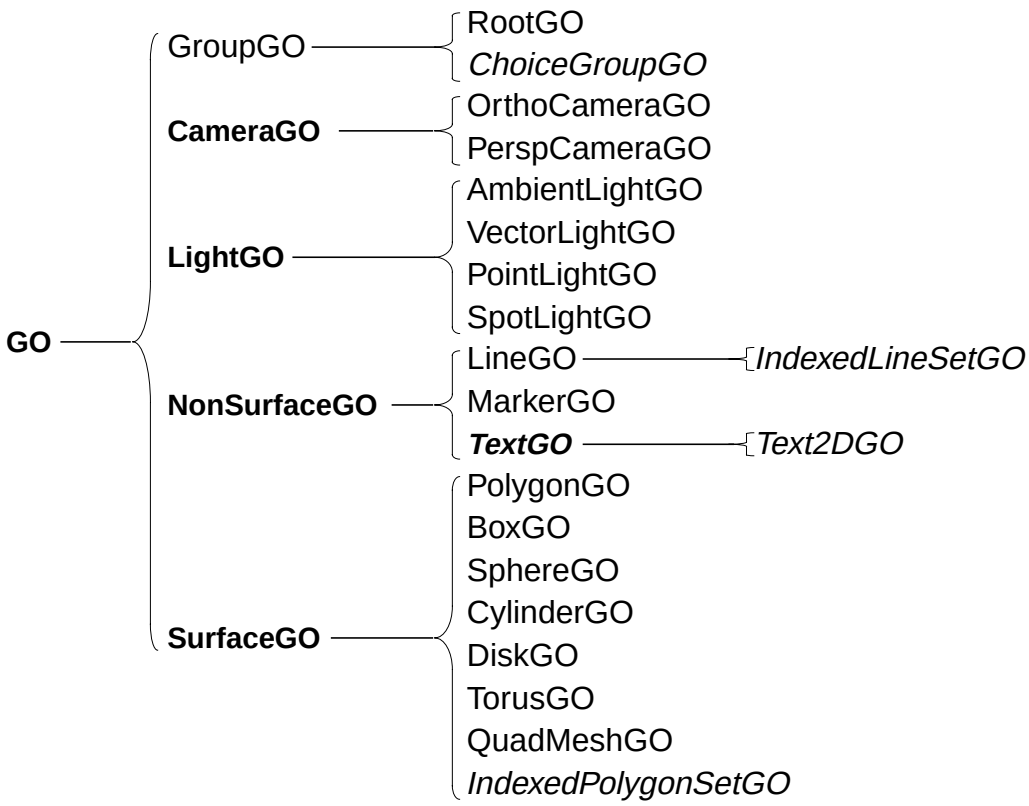


Figure 5-2: The Repo-3D GO class hierarchy. Most of the classes are also in Obliq-3D. The italicized ones were added to Repo-3D. The bold classes are abstract.

do not start satisfying their requests until the animation handle is signalled. By linking multiple properties to the same handle, a set of property value changes can be synchronized.

Associated with each GO *g* is a partial mapping of property names to values determined by the properties that have been associated with *g*. A property associated with *g* affects not only *g* but all the descendants of *g* that do not override the property. A single property may be associated with any number of GOs. It is perfectly legal to associate a property with a GO that is not affected by it; for example, attaching a “Surface Color” property to a GroupGO does not affect the group node itself, but could potentially affect the surface color of any GO contained in that group. A RootGO sets an initial default value for each named property.

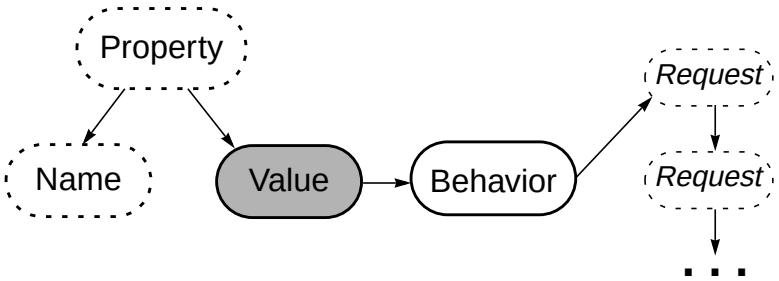


Figure 5-3: The relationship between properties, names, values, and behaviors. Each oval represents an object and arrows show containment.

There are three types of input event callbacks in Anim-3D, corresponding to the three kinds of interactive events they handle: *mouse* callbacks (triggered by mouse button events), *motion* callbacks (triggered by mouse motion events) and *keyboard* callbacks (triggered by key press events). Each object has three callback stacks, and the interactive behavior of an object can be redefined by pushing a new callback onto the appropriate stack. Any event that occurs within a root window associated with a RootGO *r* will be delivered to the top handler on *r*'s callback stack. The handler could delegate the event to one of *r*'s children, or it may handle it itself, perhaps changing the graphical scene in some way.

DistAnim-3D is the Modula-3 3D graphics and animation library underneath Repo-3D. It is a direct descendant of Anim-3D in which many of the graphical objects are distributed by being implemented with the Shared Object package. In addition to the objects being distributed, it has had many additional facilities added to it to support general-purpose 3D graphical applications, which are discussed in Appendix G. These include the addition of new GOs supporting indexed line and polygon sets, choice groups, and text (as shown in Figure 5-2), plus new properties to support these new nodes (such as font name or text style) and to enhance existing GOs with features such as texture mapping. DistAnin-3D also includes a new pair of callbacks (*projection* and *transformation* callbacks) and support for 2D picking.

Since DistAnim-3D is embedded in Repo instead of Obliq (see Chapter 4), the resulting library is called Repo-3D. The interfaces for all of Repo-3D's modules can be

found in Appendix H. In the rest of this chapter, we will refer to either DistAnim-3D or Repo-3D, as appropriate.

5.3 Design Of Repo-3D

Repo-3D's design has three logical parts: the conversion to Shared Objects, the introduction of local variations, and support for extensibility. These are the topics of Sections 5.3.1 through 5.3.3. Local variations are introduced to handle two issues mentioned in Section 5.1: transient local changes and responsive local editing.

5.3.1 Conversion to Shared Objects

The Anim-3D scene-graph model is well suited for adaptation to a distributed environment. First, in Anim-3D, properties are attached to nodes, not inserted into the graph, and the property and child lists are unordered (i.e., the order in which properties are assigned to a node, or children are added to a group, does not affect the final result). In libraries that insert properties and nodes in the graph and execute the graph in a well-defined order (such as Inventor), the *siblings* of a node (or subtree) can affect the attributes of that node (or subtree). In Anim-3D, and similar libraries (such as Java 3D), properties are only inherited *down* the graph, so a node's properties are a function of the node itself and its ancestors—its siblings do not affect it. Therefore, subtrees can be added to different scene graphs, perhaps in different processes, with predictable results.

Second, the interface (both compiled Anim-3D and interpreted Obliq-3D) is programmatic and declarative. There is no "graphical scene" file format per se: graphical scenes are created as the side effect of executing programs that explicitly create objects and manipulate them via the object methods. Thus, all graphical objects are stored as the Repo-3D programs that are executed to create them. This is significant, because by using the Shared Object package to make the graphical objects distributed, the "file format" (i.e., a Repo-3D program) is updated for free.

Converting Anim-3D objects to Shared Objects involved three choices: what objects should be replicated using Shared Objects, what methods update the state of those

objects, and what the global, replicated state of each of those objects is. Since Shared Objects have more overhead (e.g., method execution time, memory usage, and latency when passed between processes), not every category of object in Repo-3D is replicated using them. We will consider each of the object categories described in Figure 5.2 in turn: graphical objects (GOs), properties (values, names, behaviors), animation handles, and input callbacks. For each of these objects, the obvious methods are designated as update methods, and, as discussed in Chapter 3, the global state of each object is implicitly determined by those update methods. After discussing those three classes of objects, Repo-3D's support for change notification will be discussed.

5.3.1.1 Graphical Objects

GOs are the most straightforward to address. There are currently twenty-one concrete types of GOs, as shown in Figure 5-2, and all but the RootGOs are replicated. For complete details of all the Repo-3D GOs, see Appendix H.1. Since RootGOs are associated with an onscreen window, they are not replicated—window creation remains an active decision of the local process. Furthermore, if replicated windows are needed, the general-purpose programming facilities of Repo can be used to support this in a relatively straightforward manner.

A GO's state is comprised of the properties attached to the object (manipulated by the methods `setProp`, `getProp` and `unsetProp`), its name (manipulated by the methods `setName`, `getName` and `findName`), and some other non-inherited property attributes.[1] The update methods are those that modify the properties (`setProp` and `unsetProp`) or change the name (`setName`). The class definition for the base GO class is shown in Figures 5-10 and 5-11, and an example of a concrete GO (`BoxGO`) is shown in Figure 5-12.

---

1. Some attributes of a GO, such as the arrays of Point3D properties that define the vertices of a polygon set, are not attached to the object, but are manipulated through method calls. This was an outgrowth of the original Obliq-3D design that we decided not to change because the benefit of doing so would be small compared to the implementation effort.

Group GOs also contain a set of child nodes, and have additional update methods that modify that set (`add`, `remove`, `replace`, `flush` and `content`). Each of these methods, except `content`, is an update method. The class definition for the Group GO class is shown in Figure 5-13.

5.3.1.2 Properties

Properties are more complex. There are far more properties in a graphical scene than there are graphical objects, they change much more rapidly, and each property is constructed from a set of Modula-3 objects. There are currently 101 different properties of seventeen different types in Repo-3D, and any of them can be attached to any GO. For complete details of all the Repo-3D properties, see Appendix H.2. A typical GO would have anywhere from two or three (e.g., a BoxGO would have at least two properties to define its corners) to a dozen or more. And, each of these properties could be complex: in the example in Section 5.4, a single synchronous property for a long animation could have hundreds of requests enqueued within it.

Consider again the object structure illustrated in Figure 5-3. A property is defined by a name and a value, with the value being a container for a behavior. Only one of the Modula-3 objects is replicated using Shared Objects, the property *value.* Property values serve as the replicated containers for property behaviors. To change a property, a new behavior is assigned to its value via `setBeh`, which is a property value's only update method. The state of the value is the current behavior.

The other Modula-3 objects that make up a property are not replicated using Shared Objects, for the following reasons:

- *Properties* represent a permanent binding between a property value and a name. Since they are immutable, they have no synchronization requirements and can simply be copied between processes.
- *Names* represent simple constant identifiers, and are therefore also replicated by simple copying.
- *Behaviors* and *requests* are not replicated. While they can be modified after being created, they are treated as immutable data types for two reasons. First, the vast majority of

behaviors, even complex synchronous ones, are not changed once they have been created and initialized. Thus, there is some justification for classifying the method calls that modify them as part of their initialization process. The second reason is practical and much more significant. Once a scene has been created and is being "used" by the application, the bulk of the time-critical changes to it tend to be assignments of new behaviors to the existing property values. For example, an object is moved by assigning a new (often constant) behavior to its `GO_Transform` property value. Therefore, the overall performance of the system depends heavily on the performance of property value behavior changes. By treating behaviors as immutable objects, they can simply be copied between processes without incurring the overhead of the replicated object system.

5.3.1.3 Animation Handles

Animation handles are also replicated using Shared Objects. They tie groups of related synchronous properties together, and are the basis for the interaction in the example in Section 5.4. For the details of the Repo-3D animation handle modules, see Figure 5-14 and Appendix H.3. In Anim-3D, handles have one `animate` method, which starts an animation and blocks until it finishes. However, since update methods are executed everywhere, and block access to the object while they are being executed, they should not take such an extended period of time. Therefore, in Repo-3D the `animate` method is a non-update method that simply calls two new methods in sequence: an update method that starts the animation and returns immediately (`startAnimation`), and a non-update method that waits for the animation to finish (`finishAnimation`).

Repo-3D animation handles also include methods to pause (`pauseAnimation`) and resume (`continueAnimation`) an animation, to retrieve (`getAnimationTime`) and change (`gotoAnimationTime`) the current relative time of an animation handle, to retrieve the length of the animation (`getAnimationLength`), and to stop an animation early (`stopAnimation`). The global state of an Animation handle is two boolean values that indicate if it is active and/or paused or not, plus two real values corresponding to the start and current times of the animation.

Aside from these changes, there is another important difference between animation handles in Obliq-3D and Repo-3D, resulting from the distributed context. In Obliq-3D, an animation handle "finishes" (i.e., the `finishAnimation` method returns in that process) when it reaches the end of the animation, which is defined to be the time at which all of the synchronous properties attached to the handle have finished their animations. In Repo-3D, since animation handles might be replicated across a set of processes, and individual properties are only distributed to processes that need them, it is unlikely that exactly the same set of synchronous properties will exist, and thus be associated with the handle, in all processes. This implies that the finishing time for the set of synchronous properties associated with an animation handle may vary across the replicas of that animation handle.

Therefore, when an animation handle is signalled, there are two options for what time to use as the "end" of the animation: when the local properties finish their animations (the *local* time), or when all properties attached to the animation handle in all processes finish their animations (the *global* time). If we choose to enforce a common global end time, that end time may be significantly different at each replica than the length of the synchronous properties at that site, causing the animation handle not to "finish" until before or after the local animations have completed. Furthermore, there are other issues to consider if we use a uniform global end time, such as whether or not to count the time used by local property variations in the "total time" of the animation.[2]

We opted for the local solution, where the animation handle is considered finished in some process when the synchronous properties that exist in that process have finished their animations. We chose this option for three reasons. First, this was by far the simplest solution to implement and easiest for programmers to understand. Second, this useful

---

2. There are other choices, such as having all the replicas of the animation handle use the local animation length of the replica at the site that signaled the animation handle. This would allow all sites to see a uniform ending time, without needing to determine the global animation length. However, this option has the main disadvantage of the global time (the animation handle "finishes" in most processes at a different time than the local animations finish) with the further disadvantage that the animation handle's end time in most processes is meaningless with respect to the animations in that process. Therefore, we decided not to use options such as this.

piece of information (when are the animations in the local process finished?) would be hard to obtain in any other way. Finally, if a programmer needs to know when the animations have finished at all sites, they can use Repo's general programming facilities to implement such a feature in an application specific way.

5.3.1.4 Input Callbacks

In Repo-3D, input event callbacks are not replicated. As discussed in Section 5.2, input events are delivered to the callback stacks of a RootGO. Callbacks attached to any other object receive input events only if they are delivered to that object by the programmer, perhaps recursively from another input event callback (such as the one attached to the RootGO). Therefore, the interactive behavior of a root window is defined not only by the callbacks attached to its RootGO, but also by the set of callbacks associated with the graph rooted at that RootGO. Since the RootGOs are not replicated, the callbacks that they delegate event handling to are not replicated either. If a programmer wants to associate callbacks with objects as they travel between processes, Repo's general-purpose programming facilities can be used to accomplish this in a straightforward manner. For the details of the Repo-3D input callback modules, see Appendix H.4.

5.3.1.5 Change Notification

The final component of Repo-3D is support for notification of changes to distributed objects. For example, when an object's position changes or a new child is added to a group, some of the processes containing replicas may wish to react in some way. Fortunately, as discussed in Chapter 3, the Shared Object package automatically generates Callback Object types for each replicated object type, which provide the required functionality.

The Callback Objects are exposed into Repo via three modules: `AnimHandleCB`, `GOCB` and `PropCB`. The animation handle Callback Object is exposed directly into Repo via the `AnimHandleCB` module, and is used analogously to the Modula-3 object (see Appendix H.3.2). Unlike the animation handle Callback Object, the multitude of Callback Objects for the various GO and property value Shared Objects are not exposed into individual modules; while each has a separate Callback Object generated for it, they are

```
   PropCB_New(obj: Prop, overrides: Obj): T;
   PropCB_Cancel(cbobj: T): T;
WHERE
   T <: {simple} & overrides;
   overrides contains one or more of these callback methods:
      pre`init(obj: Prop, beh: PropBeh): bool;
      post`init(obj: Prop, beh: PropBeh): bool;
      pre`setBeh(obj: Prop, beh: PropBeh): bool;
      post`setBeh(obj: Prop, beh: PropBeh): bool;
      pre`anyChange(obj: Prop);
      post`anyChange(obj: Prop);

   Where Prop is a Property and PropBeh is a Property Behavior of
   the appropriate types
```

(a) The Repo help for the `PropCB` module.

```
   GOCB_New(obj: GO, overrides: Obj): T;
   GOCB_Cancel(cbobj: T): T;
WHERE
   T <: {simple} & overrides;
   overrides contains one or more of these callback methods:
      pre`propagateLocalProps(obj: GO, add del: [Prop_T]): Bool
      post`propagateLocalProps(obj: GO, add del: [Prop_T]): Bool
      pre`setProp(obj: GO, prop: Prop_T): Bool
      post`setProp(obj: GO, prop: Prop_T): Bool
      pre`unsetProp(obj: GO, name: Prop_Name): Bool
      post`unsetProp(obj: GO, name: Prop_Name): Bool
      pre`setName(obj: GO, name: Text): Bool
      post`setName(obj: GO, name: Text): Bool
      pre`anyChange(obj: GO);
      post`anyChange(obj: GO);
   If T is GroupGO or ChoiceGroupGO overrides may also contain:
      pre`add(obj new: GO): Bool
      post`add(obj new: GO): Bool
      pre`remove(obj old: GO): Bool
      post`remove(obj old: GO): Bool
      pre`replace(obj old new: GO): Bool
      post`replace(obj old new: GO): Bool
      pre`flush(obj: GO): Bool
      post`flush(obj: GO): Bool
      pre`propagateLocalChildren(obj: GO, add remove: [GO]): Bool
      post`propagateLocalChildren(obj: GO, add remove: [GO]): Bool

   ... and so on for object specific methods of other GOs ...
```

(b) Excerpts from the Repo help for the `GOCB` module.
The type of the GO determines which methods of the `overrides` objects will actually be used, and what parameters they should have.

Figure 5-4: The `GOCB` and `PropCB` modules. There are two commands in each module, one to create a new callback and another to cancel an existing one. The `overrides` parameter is a simple object containing the callback methods.

merged into the GOCB (for notification of changes to GOs) and PropCB (for notification of changes to property values) modules. This is primarily done for simplicity, since programmers frequently want to be notified of some change that is independent of the type of the GO or property value (e.g., the attachment of a new property to the object, or the assignment of a new behavior to a property value).

Hiding the property value callbacks inside PropCB is straightforward, since all of the property value Callback Objects have exactly the same set of methods, differing only in the type of their parameters (as shown in Figure 5-4(a) and Appendix H.2.2). Similarly, the GO callback objects share many methods because most of the commonly used update methods are part of GO.T, the root of the object hierarchy (as shown in Figure 5-4(b)). However, many GOs have additional update methods, so the GOCB module is more complex; the additional callback methods for group GOs are shown in Figure 5-4(b). (The complete specification of the GOCB module can be found in Appendix H.1.2.)

By integrating these change notification callbacks into a pair of modules, a programmer need not know the specific kind of GO or Property for which they are creating a Callback Object, as long as the object is a subtype of the one they are expecting; the wrapper modules look at the type of the GO or Property and create the appropriate type of Callback Object.

5.3.2 Local Variations

Repo-3D's *local variations* solve a set of problems particular to the distributed context in which Repo-3D lives: maintaining interactivity and supporting local modifications to the shared scene graph.

If the graphical objects and their properties were always strictly replicated, programmers would have to create local variations by copying the objects to be modified, creating a set of Callback Objects on the original objects, the copies of those objects, and all their properties (to be notified when either change), and reflecting the appropriate changes between the instances. While this process could be automated somewhat, it would still be tedious and error prone. More seriously, the overhead of creating this vast array of objects

and links between them would make this approach impractical for short transient changes, such as highlighting an object under the mouse.

To overcome this problem, Repo-3D allows the two major elements of the shared state of the graphical scene—the properties attached to a GO and the children of a group—to have *local variations* applied to them. Local variations on property values or animation handles are not currently supported.

Conceptually, local state is the state added to each object (the additions, deletions, and replacements to the properties or children) that is only accessible to the local copies and is not passed to remote processes when the object is copied to create a new replica. The existence of local state is possible because the shared state of a replicated object is not a function of the data elements of the object, but is instead defined implicitly by the methods that update the object, as discussed in Chapter 3. Therefore, since the new methods that manipulate the local variations do not modify the shared state, they are added to the GOs as *non-update* methods; notice that in Figure 5-13, none of the GO or group GO local variation methods are denoted as update methods in the SHARED UPDATE METHODS pragma. Repo-3D combines both the global and local state when creating the graphical scene using the underlying graphics package. Repo-3D ensures that the local state is not copied when an object is first passed to a new process by defining custom pickling routines that pass the global state and initialize the local state on the receiving side to empty values (recall the discussion of custom pickling routines in Section 3.4.1.3).

As mentioned above, local variations come in two flavors:

- *Property variations.* There are three methods to set, unset, and get the global property list attached to a GO. We added the following methods to manipulate local variations: add or remove local properties (overriding the value normally used for the object), hide or reveal properties (causing the property value of the parent node to be inherited), and flush the set of local variations (removing them in one step) or atomically apply them to the global state of the object. See Figure 5-10 for the specification of the Modula-3 GO object that includes these methods, and Appendix H.1.1 for the Repo-3D module interface.

- *Child variations.* There are five methods to add, remove, replace, retrieve, and flush the set of children contained in a group node. We added the following ones: add a local node, remove a global node locally, replace a global node with some other node locally, remove each of these local variations, flush the local variations (remove them all in one step), and atomically apply the local variations to the global state. See Figure 5-13 for the complete specification of the Modula-3 group GO object that includes these methods, and Appendix H.1.10 for the Repo-3D module interface.

This set of local operations supports the problems local variations were designed to solve, although some possible enhancements are discussed in Section 7.1.

5.3.3 Extensibility

Repo-3D objects are extensible so that the application state can be added to the Repo-3D objects, allowing programmers to avoid the dual database problem (as discussed at the beginning of the chapter). Objects can be extended at both the compiled and interpreted levels of Coterie. At the Modula-3 level, DistAnim-3D objects can be subtyped and extended to create new Shared Objects, as discussed in Chapter 3.

Programmers can extend the Repo-3D objects as well, but care must be taken because of the way DistAnim-3D's replicated objects are embedded in Repo. In particular, programmers must keep two restrictions in mind when extending Repo-3D objects. First, Repo-3D objects cannot be extended like other Repo objects (by cloning them, as discussed in Section 4.4.2). Instead, programmers must use a Repo-3D object's `extend` method for this purpose. Second, the Repo-3D object that exposes the DistAnim-3D replicated object into Repo is a `simple` (unsynchronized replicated) Repo object, so if a data field is to be changed after an object is distributed, the data field itself must be a synchronized replicated object or the changes will not be distributed to all replicas.

To understand where these restrictions come from, we need to explain the way in which the DistAnim-3D objects are embedded in Repo (shown in Figure 5-5). The Repo object that exposes a DistAnim-3D object into Repo has methods corresponding to the various methods of the DistAnim-3D object, and a single data field (`raw`) that points at

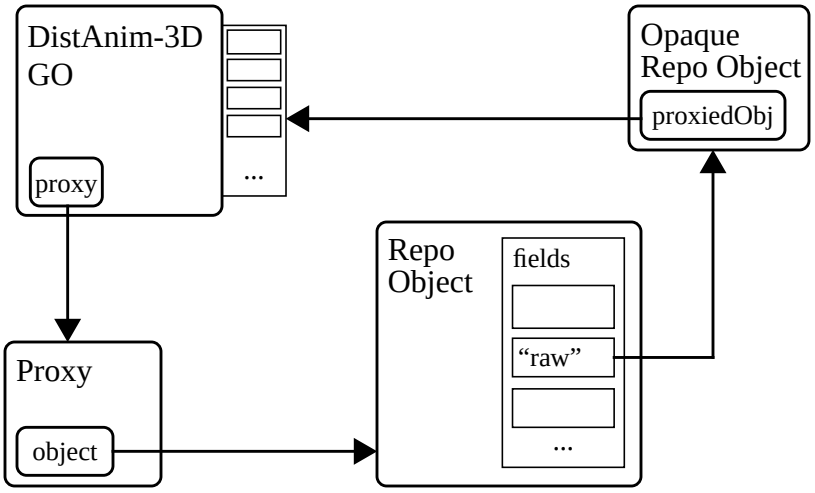


Figure 5-5: Embedding DistAnim-3D objects in Repo. Anim-3D (and, therefore, DistAnim-3D) was designed to be embedded in an interpreted language, which in our case is Repo. Therefore, each DistAnim-3D object (i.e., GOs, properties, behaviors, etc.) has a `proxy` field that will point to a `Proxy` object when the representation of the object in Repo (the "proxy") has been created (this proxy can be created when the graphical object is created, or deferred until the object is accessed from Repo). The `Proxy` object contains an untyped reference field `object` that points at the Repo wrapper object. To allow the DistAnim-3D object to be retrieved from the Repo object, an opaque Repo type is created for every type of graphical object. This object contains a single field, `proxiedObj`, that points at the original graphical object. An instance of the appropriate type is assigned to a field named "`raw`" in the Repo object.

the underlying DistAnim-3D object. The choice of whether a given object (i.e., a GO, property value, property behavior, etc.) is synchronized or unsynchronized is made at the DistAnim-3D level, and all of the Repo objects are unsynchronized (regardless of the type of the underlying DistAnim-3D object).

The primary reason for having the Repo wrapper object unsynchronized is as follows: if both the Repo and DistAnim-3D objects were synchronized, invoking an update method in the wrapper object would result in that method being invoked everywhere, but the action taken at every site by that method would typically include invoking a method on the underlying DistAnim-3D object *at every site*. Each invocation of the DistAnim-3D method would cause the method to be invoked on all replicas. The result would be to have the DistAnim-3D method invoked N^2 times (if there are N replicas), N times for each of the N invocations of the Repo update method.

The structure of the Repo-3D object in Figure 5-5 is also the reason that Repo-3D objects should only be extended to contain additional fields by calling the object's `extend` method. This method takes a single argument, an object containing the fields to be added to the Repo-3D object. The method creates a new Repo object combining the old Repo object and these new fields, and updates the `Proxy` object's `object` field to point at this new Repo object. If the Repo object were extended using Repo's `clone` operator, the internal Proxy object would continue to point at the original Repo object, breaking the circular reference structure. This structure also implies that Repo-3D objects should only be extended before they are first passed out of their original process, because the structure would only be updated at the local site.

The final implication of the way DistAnim-3D objects are embedded in Repo arises because the Repo objects are unsynchronized. If a programmer wishes to add application data fields to the Repo-3D objects, and then distribute the Repo-3D object around the network, subsequent changes to the application data fields will not be propagated globally because the Repo objects are not synchronized. To get around this limitation, the programmer should add synchronized data fields to these unsynchronized Repo-3D objects. These replicated data objects would be pulled around the network with the Repo-3D object, and any changes made to those data values would be propagated. See the example in Section 6.2 for an illustration of extending objects in this way.

5.4 Examples

In this section we will give a number of examples that demonstrate the utility of Repo-3D. First, we will give a simple tutorial example that shows how to build a simple distributed graphical scene. In Section 5.4.2, we will return to the tracker report distribution example introduced in Section 2.5.1, and discuss how tracker data can be transparently distributed using Repo-3D properties.

Next, we will show how the ability to embed application state in replicated objects neatly solves the dual database problem, using an example of the creation of a new Truncated Pyramid GO taken from our integration of the Brown Sketch system into Coterie (this integration will be discussed further in Section 6.2). Finally, in Section 5.4.4 we will

discuss the design of a distributed animation viewer, and show how local variations address a number of common problems in distributed, collaborative applications.

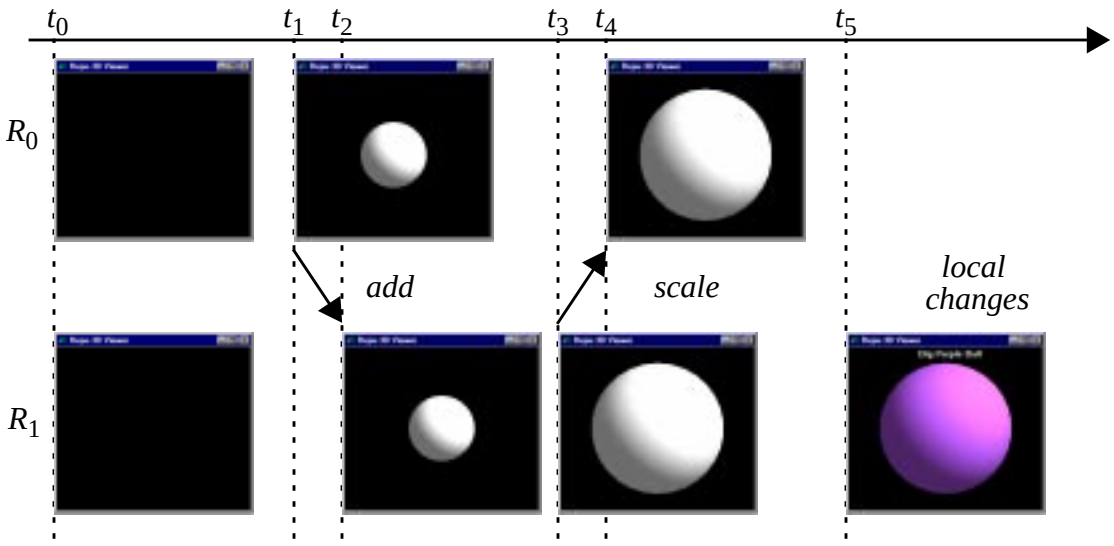
5.4.1 A Tutorial Example

The first example we will give, shown in Figure 5-6, highlights the simplicity of using Repo-3D to create a distributed graphical scene. In this example, two processes are running, each containing a Repo-3D root GO. One process creates a group GO and exports it to the network, and the other process imports it. When both link this group GO to their root GO, they now have a symmetric, shared graphical scene. Any changes made to one are reflected in the other (e.g., adding an object, such as the sphere in the figure).

While one could imagine writing a simple distributed data structure to mimic this behavior, by sharing a group of objects and changing the local root GO to reflect changes to the contents of this shared data structure, realistic applications require more complex data sharing, which Repo-3D also provides in a straightforward manner. For example, if one wanted to ensure the objects in our hypothetical shared data structure were accurately reflected in our local root GO, we would also have to watch for all possible changes to the properties of those objects and apply those changes locally. Since Repo-3D's objects are fully replicated, it provides this behavior transparently. Similarly, if the objects being shared were more complex than the simple sphere in the example, perhaps containing a complex hierarchy of group GOs, the programmer creating our hypothetical shared data structure would have to watch the entire hierarchy for changes. Repo-3D provides this behavior for free as well.

5.4.2 Yet Another Tracker Example

Now, we shall return to the tracker distribution example introduced in Section 2.5.1. A simple tracker distribution object was implemented as a Modula-3 Shared Object in Section 3.4.1, and reimplemented a number of different ways using Repo replicated objects in Section 4.6.1. Repo-3D properties provide a simple way of distributing the position and orientation of a tracker, which is more appropriate than the approaches discussed in previ-



(a) The timeline of the simple animation, showing the two windows at different times, with arrows indicating data flow between the processes.

```
let r = RootGO_NewStd();
let g = GroupGO_New();
net_export("g", host, g);
r.add(g);
```

```
let r = RootGO_NewStd();
let g = net_import("g", host);
r.add(g);
```

(b) The code executed at time=t_0, in window R_0 (left) and R_1 (right)

```
let s = SphereGO_New([0,0,0],1);
g.add(s);
```

(c) The code executed in window R_0 at time=t_1

```
GO_SetTransform(g,Matrix4_Scale(Matrix4_Id, 2,2,2));
```

(d) The code executed in window R_1 at time=t_3

```
s.setLocalProp(SurfaceGO_Color,ColorProp_NewConst("purple"));
let t = Text2DGO_New([0,1.1,0], "Big Purple Ball", "Center");
g.localAdd(t);
```

(e) The code executed in window R_1 at time=t_5

Figure 5-6: A simple Repo-3D example. In this example, for which all the code is shown, we have two windows (R_0 and R_1) in two separate processes. The two windows are initialized at time t_0, as shown, so that they contain a shared group GO. If either processes changes the GO, the changes will be reflected in both. Therefore, when a sphere is added to R_0 (at time t_1), the update is distributed and applied in the process containing R_1 (at time t_2). This sphere is also shared; when it is scaled in R_1 (at time t_3), it is also scaled in R_0 (at time t_4). However, local updates can be applied to either without requiring network traffic: when the color of the sphere is locally changed and a 2D text object locally added in R_1 (at time t_5), these changes are not sent to, or reflected in, R_0.

ous chapters in certain situations, such as when the position and orientation of the tracker are only used to position objects in the graphical scene.

In those cases, Repo-3D properties can be used to transparently distribute the tracker data. Instead of setting the value of a shared tracker position object, the process reading and processing the tracker can create Repo-3D constant transformation property behaviors, and assign each new behavior to one or more transformation properties. These properties can be attached to any number of GOs and distributed to any number of processes, and those GOs will transparently follow the tracker.

5.4.3 A Truncated Pyramid Object

One of the common things to do when building graphical applications is to create new domain-specific objects that can be treated like the built-in graphical objects, but contain addition semantic information specific to that domain. In this section, we will present one such object, a Truncated Pyramid, that was created for the Sketch example presented in Section 6.2. In that example, we created four new Repo-3D, objects representing Truncated Pyramids and Cones, Extrusions and Surfaces of Revolution. While the Truncated Pyramid is quite simple, it serves as an example of how to extend Repo-3D objects to contain application specific state.

A Truncated Pyramid is defined as follows. The pyramid is centered around the origin, with its bottom face being a square from (–1,–1,–1) to (1,–1,1). The top face of the pyramid can be of any size, but will always lie in the Y=1 plane. The *taper* vector determines how much the top face of the pyramid should be tapered in from the default position of (x=1, z=1) and (x=–1, z=–1). The *shear* vector determines how much the center of the top face of the pyramid should be offset from the Y axis. We define the Truncated Pyramid as an extension to the `IndexedPolygonSet` object, as shown in Figure 5-7. By using a replicated object to contain the taper and offset information (as discussed in Section 5.3.3), and defining the polygon set using asynchronous point properties that reference the fields of this object, the offset and taper values of the truncated pyramid object can be modified at any time by simply changing the corresponding field of this replicated object.

```
module TruncPyrGO;
let New = proc (taper, offset)
     let obj = {replicated, offset => simple(offset),
                taper => simple(taper)};
     IndexedPolygonSetGO_NewWithShapeHint(
        [[-1.0,-1.0,-1.0], [ 1.0,-1.0,-1.0],
         [ 1.0,-1.0, 1.0], [-1.0,-1.0, 1.0],
         PointProp_NewAsync(meth (s,t)
             [(obj.offset[0] - obj.taper[0]), 1.0,
              (obj.offset[1] - obj.taper[1])]
          end),
         PointProp_NewAsync(meth (s,t)
             [(obj.offset[0] + obj.taper[0]), 1.0,
              (obj.offset[1] - obj.taper[1])]
          end),
         PointProp_NewAsync(meth (s,t)
             [(obj.offset[0] + obj.taper[0]), 1.0,
              (obj.offset[1] + obj.taper[1])]
          end),
         PointProp_NewAsync(meth (s,t)
             [(obj.offset[0] - obj.taper[0]), 1.0,
              (obj.offset[1] + obj.taper[1])]
          end)],
        [[0,1,2,3], [7,6,5,4], [1,0,4,5],
         [2,1,5,6], [3,2,6,7], [0,3,7,4]],
        1.57, "Convex").extend({simple, data => obj});
  end;
end module;

addhelp TruncPyrGO short "A truncated pyramid GO" full
"  TruncPyrGO_New(taper: Point2, offset: Point2):
       IndexedPolygonSetGO
";
```

Figure 5-7: The `TruncPyr` object. The Truncated Pyramid object is an indexed polygon set with eight vertices and six faces. The bottom four vertices are constants, defining the square from [–1,–1,–1] to [1,–1,1]. The top four vertices are asynchronous point properties, that compute their values using the current values of the offset and taper fields of the replicated object `obj`. The object `obj` is added to the indexed polygon set object as a new field, `data`. When either the taper or offset field of the replicated object is changed, the changes are reflected in all replicas and the indexed polygon set immediately changes its appearance to reflect the new values.

5.4.4 An Animation Examiner

A more complex and complete example of prototyping distributed applications with Repo-3D is the distributed animation examiner we created for the CATHI animation generation system. CATHI generates short informational animation clips to explain the operation of

technical devices [Butz, 1997]. The scripts it generates describe full-featured animations, including camera and object motion, color and opacity effects, and lighting setup.

It was reasonably straightforward to modify CATHI to generate Repo-3D program files. The Repo-3D program creates two scene graphs: a camera graph and an animation scene graph. The objects in these graphs have synchronous behaviors specified for their surface and transformation properties. An entire animation is enqueued in the requests of these behaviors, and can last anywhere from a few seconds to a few minutes.

We built a distributed, multi-user examiner for these animations over the course of a weekend. The examiner allows multiple users to view the same animation while discussing it (e.g., via electronic chat or on the phone). Figure 5-8 shows images of the examiner running on four machines, each with a different view of the scene. The first step was to build a simple “loader” that reads the animation file, creates a root GO, adds the animation scene and camera to this GO, and exports the animation to the network. This required ten lines of Repo-3D code. A “network” client, which imports the animation from the network instead of reading it from disk, replaced the two lines of code to read and export the animation with a single line to import it, but was otherwise identical to the loader program. Figure 5-8(a) shows an animation being viewed by one of these clients.

An animation examiner program is loaded by both these simple clients, and is about 450 lines long. The examiner supports:

- Pausing and continuing the animation, and changing the current animation time using the mouse. Since this is done by operating on the shared animation handle, changes performed by any viewer are seen by all. Because of the consistency guarantees, all users can freely attempt to change the time, and the system will maintain all views consistently.
- Opening and closing a second “overview” window (Figure 5-8(b)), where a new camera watches the animation scene and camera from a distant viewpoint. A local graphical child (representing a portion of the animation camera’s frustum) is added to the shared animation camera group to let the attributes of the animation camera be seen in the overview window.

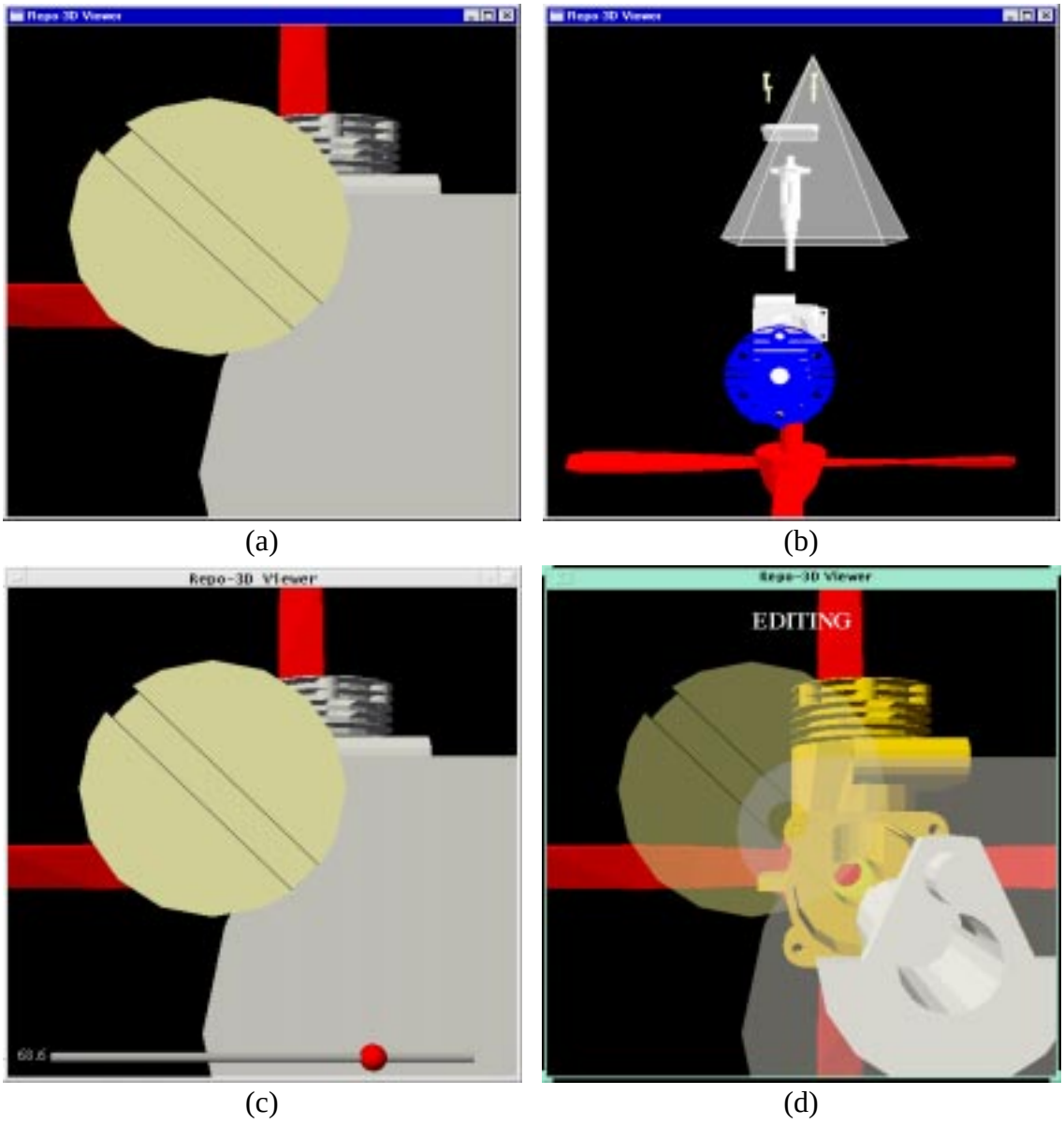


Figure 5-8: The distributed CATHI animation viewer. Simultaneous images from a session with the viewer, running on four machines, showing an animation of an engine. (a) Plain animation viewer, running on Windows NT. (b) Overview window, running on Windows 95. (c) Animation viewer with local animation meter, running on IRIX. (d) Animation viewer with local transparency to expose hidden parts, running on Solaris.

- A local animation meter (bottom of Figure 5-8(c)), that can be added to any window by pressing a key, and which shows the current time offset into the animation both graphically and numerically. It is added in front of the camera in the animation viewer window, as a local child of a GO in the camera graph, so that it is fixed to the screen in the animation viewer.

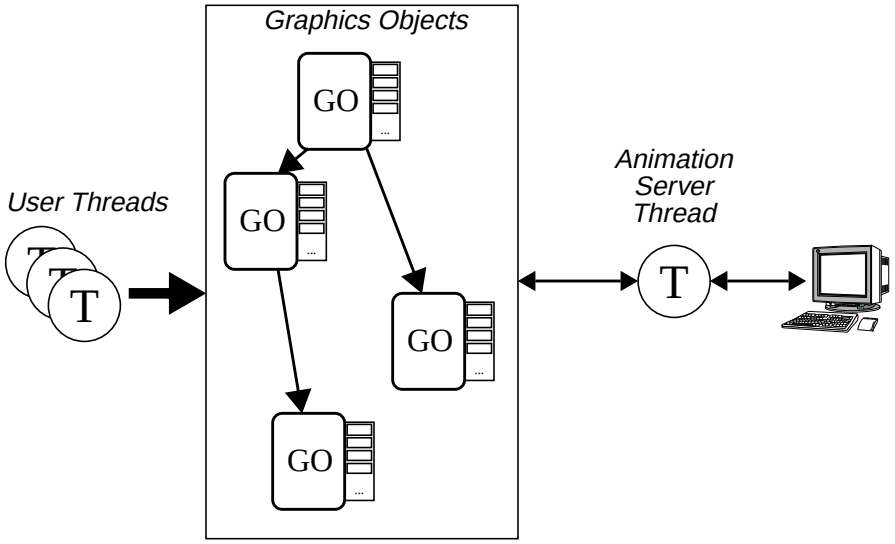
- Local editing (Figure 5-8(d)), so that users can select objects and make them transparent (to better see what was happening in the animation) or hide them completely (useful on slow machines, to speed up rendering). Assorted local feedback, such as highlighting the object under the mouse and flashing the selected object, is done with local property changes to the shared GOs in the scene graph.

Given the attention paid to the design of Repo-3D, it was not necessary to be overly concerned with the distributed behavior of the application (we spent no more than an hour or so). Much of that time was spent deciding if a given operation should be global or a local variation. The bulk of programming and debugging time was spent implementing application code. For example, in the overview window, the representation of the camera moves dynamically, based on the bounding values of the animation's scene and camera graphs. In editing mode, the property that flashes the selected node bases its local color on the current global color (allowing a user who is editing while an animation is in progress to see any color changes to the selected node.)

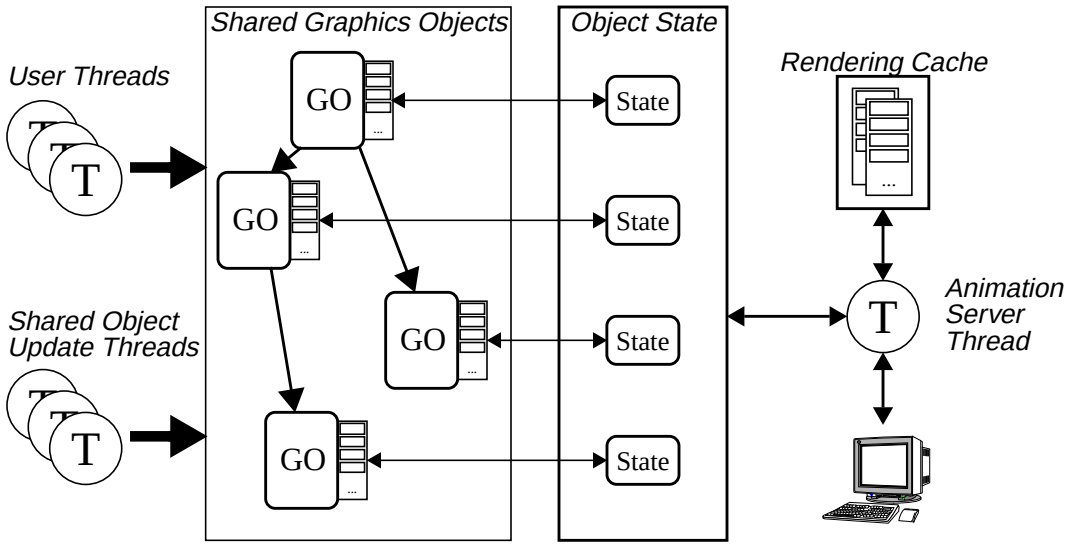
5.5 Implementation

In this section, we will discuss the implementation of DistAnim-3D's shared graphical objects, highlighting the changes made to Anim-3D during the move to a distributed context based on the Shared Object programming model.

The differences in the internal structure of Anim-3D and DistAnim-3D result from both the rendering optimizations (discussed in Appendix G), and the conversion to Shared Objects, shown in Figure 5-9. While the data structures have changed considerably, the basic structure of the code has not; user threads update the graphics objects, and there is an *animation server thread* that is responsible for rendering the graphical scene and handling input. The package has two global locks, referred to as the *external lock* and the *internal lock*. Each time through its rendering loop, the animation server thread acquires the external lock and the internal lock in sequence, reacts to input from the user, handles any changes to the scene graph, and then renders the windows (if needed).



(a) The internal structure of Anim-3D. User threads and the animation server thread access the graphical objects. The animation server thread renders the scene directly from the graphical objects.



(b) The internal structure of DistAnim-3D. The user-threads access the Shared Graphical Objects, and any changes they make are immediately reflected in the State objects. The animation server thread accesses only the State objects, using them to build and update the rendering cache used to refresh the graphics display

Figure 5-9: The internal structure of Anim-3D and DistAnim-3D. The Rendering Cache was created to enhance the performance, and the Object State was separated out during the conversion to Shared Objects.

The external lock is available to programmers, and can be acquired when it is necessary to make multiple changes to the scene graph atomically; since the animation server thread needs to acquire the lock to render the scene, all changes made while the external

lock is held are atomic from the local viewers point of view. The internal lock is acquired by the graphical object methods when they update the scene graph data structures. (Two locks are needed because Modula-3 mutexes are not reentrant; if there was only one lock and the programmer acquired it to perform multiple actions atomically, the GO methods that would be called to perform those actions would deadlock the system when they tried to acquire the same lock before modifying the internal data structures.)

The straightforward part of the conversion to using Shared Objects was selecting which object methods should be update methods, as discussed in Section 5.3.1, and changing the definitions in the object class hierarchy to follow the guidelines for the Shared Object package, as discussed in Section 3.4.1. However, the Anim-3D data structures are complex, and, like most thread-safe libraries, were already designed to safely handle multiple simultaneous access to the objects through the use of the two global locks mentioned above. This means that the additional locks supplied by the Shared Object package for its objects are unnecessary to ensure the safe access to the global state. While the overhead incurred by these locks is not normally significant, in this case it is cause for concern; the animation server thread will potentially access hundreds of objects to repair and redisplay the graphical scene, and these graphical objects have over three dozen internal methods that are called repeatedly during this repair and redisplay process, with each call needing to reaquire the object's lock. Since we had previously gone to great lengths to make this repair process as efficient as possible, we want to avoid this additional overhead.

To allow the animation thread to avoid having to repeatedly acquire these locks, we moved the internal methods and all the state of the graphical objects to a second set of parallel `State` objects, as illustrated in Figure 5-9 (and can be seen in Figures 5-11 and 5-14(b)).[3] Each GO and property value now has a corresponding state object. All of the methods called by the animation server thread are in the new `State` objects, and are therefore not subject to the Shared Object locks. The methods relating to the modification

3. While we could have left the state data in the graphical objects and moved only the methods, moving both allowed us to automate the resulting modifications to the code (since both the internal method and data access now go through the state objects).

of, and access to, the global state (and only those methods) remain in the graphical objects, and are subject to the Shared Object locks.

While these modifications and optimizations required much of the code for DistAnim-3D to be touched and modified, the changes were not conceptually difficult and serve to illustrate the usefulness of the programming model. Since the Shared Object system only examines the method definitions, not the internal data, and because the package it tightly integrated with the programming language (including following the predominant Modula-3 programming style) it was straightforward to apply the model to an existing, complex software package such as Anim-3D. To further illustrate this, we will highlight some of the main objects in DistAnim-3D and show how the package fit in with the existing code.

The Anim-3D and DistAnim-3D definitions of `GO.T`, the base class for all GOs, are shown in Figure 5-10. Notice that we do not declare update methods here (using the `SHARED UPDATE METHODS` pragma), as this is an abstract type. Therefore, this object does not have the characteristic “hole” in the inheritance hierarchy (described in Section 3.4.1) that is to be filled in by the Shared Object code generator; this hole will be left in the concrete subtypes. The only changes from Anim-3D are the object inherited from, and the addition of the new methods for local property manipulation. Some of the internal details of this object are shown in Figure 5-11. The one new method defined here, `globalPropagateLocalProps`, is an update method used internally by the external `setLocalPropsGlobally` method; the external method extracts the lists of properties to be added and removed, and calls the internal method with these two lists as parameters.

This is an excellent example of the flexibility and power of the Shared Object package design, illustrating how update and non-update properties can work together: conceptually, the external method (`setLocalPropsGlobally`) is an update method, but it must do some work locally to package up the data needed for the update, since this data is local to the process where the method is invoked. The internal method

```
TYPE
   T <: Public;
   Public = ProxiedObj.T OBJECT METHODS
      ... methods ...
   END;
```

(a) The Anim-3D definition of `GO.T`. We omit the methods here. `ProxiedObj.T` is used to embed objects in an interpreted language, such as Repo or Obliq.

```
TYPE
  T <: Public;
  Public = SharedObj.T OBJECT METHODS
     init () : T;
     setName (name : TEXT);
     getName () : TEXT;
     findName (name : TEXT) : T;

     (* the original methods to manipulate properties *)
     getProp (pn : Prop.Name) : Prop.Val;
     setProp (p : Prop.T);
     unsetProp (pn : Prop.Name);                    Global Property Manipulation

     (* the methods to manipulate local properties *)
     setLocalProp (p : Prop.T);
     unsetLocalProp (pn : Prop.Name);
     getLocalProp (pn : Prop.Name) : Prop.Val;
     hideGlobalProp (pn : Prop.Name);
     revealGlobalProp (pn : Prop.Name);
     isPropHidden (pn : Prop.Name) : BOOLEAN;
     setLocalPropsGlobally ();                      Local Property Manipulation

     pushMouseCB (cb : MouseCB.T);
     popMouseCB ();
     removeMouseCB (cb : MouseCB.T);
     invokeMouseCB (mr : MouseCB.Rec);
     ... same four methods for PositionCB's and KeyCB's ...

     addProjectionCB (cb : ProjectionCB.T);
     removeProjectionCB (cb : ProjectionCB.T);
     invokeProjectionCB (READONLY pr : ProjectionCB.Rec);
     ... same three methods for TransformCB's ...
END;
```

(b) The DistAnim-3D definition of `GO.T`. The new `GO.T` inherits from `SharedObj.T` (which is a subclass of `ProxiedObj.T`).

Figure 5-10: The `GO.T` class. This is the Modula-3 base class for all DistAnim-3D GOs. In this, and all other code in this chapter, we have removed the RAISES clauses from the method and procedure declarations for clarity. We have highlighted the groups of methods used to manipulate the local and global properties.

(`globalPropagateLocalProps`) can then be called with this additional data supplied in its arguments.

```
REVEAL
  T = Public BRANDED "GO.T" OBJECT
    state: State := NIL;
  METHODS
    globalPropagateLocalProps (add, remove: PropList.T);
  OVERRIDES
    ... method overrides ...
  END;
```

Figure 5-11: Excerpts from `GOPrivate.i3`. As with `TrackerPositionF.i3`, the internal details of the `GO.T` object are exposed in this private interface. We define one additional method on the object (`globalPropagateLocalProps`), which is an update method that is used internally, and define a separate object to hold the state. The `State` object contains the local and global state, as well as methods to manipulate that state.

```
TYPE
  T <: Private;
  Private <: Public;
  <* SHARED UPDATE METHODS T.init, T.setProp, T.unsetProp,
                T.globalPropagateLocalProps, T.setName *>
  Public = SurfaceGO.T OBJECT
  METHODS
    init () : T;
  END;
```

(a) `BoxGO.T`

```
REVEAL
  Private = Public BRANDED "BoxGO.T" OBJECT
  OVERRIDES
    ... method overrides ...
  END;
```

(a) Excerpts from `BoxGOPrivate.i3`

Figure 5-12: `BoxGO.T` class definitions. Most of the GOs are similar to the `BoxGO.T` definition, and have no other methods. The internal details of the `BoxGO.T` object, including method overrides, are exposed in the private `BoxGOPrivate.i3` interface.

The definition of one of the concrete GO types, `BoxGO.T`, is shown in Figure 5-12. `BoxGO.T` is a representative example of the majority of GOs, since, like most GOs, its state is defined entirely by the properties attached to it. Therefore, no additional methods are defined (aside from an initialization method, `init()`). The `SHARED UPDATE METHODS` pragma includes all of the update methods defined in `GO.T`. Notice that, as discussed above, `globalPropagateLocalProps` is an update method, but `setLocalPropsGlobally` is not.

```
TYPE
   T <: Private;
   Private <: Public;
   <* SHARED UPDATE METHODS T.init, T.setProp, T.unsetProp,
      T.globalPropagateLocalProps, T.setName, T.add, T.remove,
      T.replace, T.flush, T.globalPropagateLocalChildren *>
   Public = GO.T OBJECT METHODS
      init (initSize := 5) : Public;

      add (o : GO.T);
      remove (o : GO.T);
      replace (new, old: GO.T);
      flush ();
      content () : REF ARRAY OF GO.T;          Global Children Manipulation

      addLocal (o : GO.T);
      removeLocal (o : GO.T);
      replaceLocal (new, old: GO.T);
      removeLocalAddition (o : GO.T);
      removeLocalRemoval (o : GO.T);
      removeLocalReplacement (old: GO.T);
      flushLocal ();
      localContent () : REF ARRAY OF GO.T;
      mergeLocalToGlobal ();                   Local Children Manipulation
   END;
```

Figure 5-13: The `GroupGO.T` class definition. This is the Modula-3 class for DistAnim-3D grouping objects. We have highlighted the groups of methods used to manipulate the local and global child lists. The `mergeLocalToGlobal` method causes the local properties to be merged into the global state, through the use of an internal update method `globalPropagateLocalChildren` (the internal details are not shown here, for brevity).

The definition of `GroupGO.T`, the class for Repo-3D group objects, is shown in Figure 5-13. This object serves as a more complex example of a Shared Object, since it inherits from `GO.T`, but also defines a new set of methods. It includes all of `GO.T`'s global and local state manipulation methods, and adds additional methods for manipulating the global and local children of a group GO. As with the `GO.T` object, one of the local methods (`mergeLocalToGlobal`) uses a private update method (`globalPropagate-LocalChildren`) to propagate the local child variations into the global state; `mergeLocalToGlobal` extracts the lists of children to be added and removed, and calls the internal method with these two lists as parameters. As can be seen in Figure 5-13, all of the update methods of `GO.T` and `GroupGO.T`, including the two private methods, are listed in the SHARED UPDATE METHODS pragma.

The definition of `Prop.Val`, the base class for all Repo-3D property values, is similar in spirit to those of `GO.T` and `BoxGO.T`. Like the GO objects, the property value objects have an internal state object so that the animation server thread can avoid the Shared Object locks. Each of the seventeen different properties is defined in a similar, straightforward way, so we will not include the details here.

The final Shared Object in DistAnim-3D is the animation handle object, `AnimHandle.T`, which is shown in Figure 5-14. This object presents another interesting example of the use of private update methods, as well as being the one component of Anim-3D that was substantially changed during the conversion to DistAnim-3D (as discussed in Section 5.3.1.3 and Appendix G). As was noted in those sections, one of the reasons we chose to have the `finishAnimation` method return when the synchronous behavior animations finish at the local site, as opposed to having it return when the animations at all sites finish, is because the implementation of the global version is difficult. The difficulty arises because of the asynchronous nature of the object updates; it would be hard to determine the current total animation length at all sites at exactly the point when the animation is signalled.

The most interesting aspect of the implementation of the animation handle object, however, is how the private update methods are used to guarantee that the distributed animations stay synchronized. Before looking at the implementation, considering the following scenario. If an animation handle is signaled at time t_0 (by having its `signal` method invoked), the animation update will not be executed until the message has returned from the sequencer, say at time t_1. In addition, the update message invoking this method will not arrive at a remote replica until some other time, say t_2. Therefore, the animation will start with a time difference of t_2-t_1 at these two sites. While t_1 and t_2 are likely to be close, they will not, in general, be equal. If the two sites have different sequencers, there will be an even greater delay caused by the additional hop on the network as the message travels between the sequencers.

There are two problems here that must be addressed: the animation starts at different times at different sites, and the animation does not start at any site at the time it is sig-

```
TYPE
   T <: Private;
   Private <: Public;
   <* SHARED UPDATE METHODS T.init, T.startAnimationTime,
      T.pauseAnimationTime, T.stopAnimation,
      T.continueAnimationTime, T.goToAnimationTimeTime *>
   Public = SharedObj.T OBJECT METHODS
      init () : T;
      startAnimation();
      finishAnimation();
      animate ();
      stopAnimation();
      pauseAnimation();
      continueAnimation();
      getAnimationTime(): LONGREAL;
      goToAnimationTime(time: LONGREAL);
      getAnimationLength(): LONGREAL;
   END;
```

(a) From `AnimHandle.i3`. The definition of the animation handle object. Notice that many of the update methods are variations of the externally visible methods, with "Time" added to their names.

```
REVEAL
   Private = Public BRANDED "AnimHandle.T" OBJECT
      state : State;
   METHODS
      startAnimationTime(time: LONGREAL) := StartAnimationTime;
      pauseAnimationTime(time: LONGREAL) := PauseAnimationTime;
      continueAnimationTime(time: LONGREAL) :=
         ContinueAnimationTime;
      goToAnimationTimeTime(time, relTime: LONGREAL) :=
         GoToAnimationTimeTime;
   OVERRIDES
      ... method overrides ...
   END;
```

(b) From `AnimHandlePrivate.i3`. Excerpts from the revelation of the `AnimHandle.Private`.

Figure 5-14: The `AnimHandle` class. Unlike the GO and property value classes, the animation handle class was changed substantially from the Obliq-3D version, which had one method, `animate`.

nalled, including the site at which the animation handle was signalled. We address both problems by using private update methods that take an additional *time* parameter that is set to the time the action was invoked at the calling site.[4] Regardless of when the update arrives and is executed at a site, the animation handle behaves as if it were signaled at the

4. In the current implementation, we assume that all of the machines have their clocks synchronized using a time-synchronization protocol such as NTP, the Network Time Protocol [Mills, 1992]

time specified by this time parameter, ensuring that the animation appears to start and stop at exactly the same time at all sites. The network delays manifest themselves by having the animation appear to "jump" when it is started or paused: because the update method will not be invoked until time t_1 in the above scenario, the animation corresponding to the time range $[t_0 ... t_1)$ will simply be skipped at the invoking site, and the range $[t_0 ... t_2)$ will be skipped at the second site. Similarly, if the `pauseAnimation` method was invoked in the same way, the animation at both sites will appear to "jump back" a small amount when it is finally invoked.

5.6 Performance

In this section, we will discuss the performance of Repo-3D. When looking at an application written using Repo-3D, it is useful to differentiate between two phases of its life cycle: the "set-up" period, when the application is building up (possibly large) shared data structures, and the "steady state" period, when the bulk of the data structures have been distributed and users are interacting with the system. Repo-3D has been optimized to perform well during the application steady state, at the expense of performance during the set-up period.

This performance trade-off can be traced back to the design of the Shared Object package. Recall from Section 3.4.2, when a Shared Object is embedded in the arguments to a Shared Object update method, only the universal identifier of the object is included in the message. This makes passing Shared Objects very efficient when the object already exists in the destination process, but slows down object transfer somewhat when the object does not yet exist. Where the application is in its steady state, and most of the Shared Objects already exist in all the processes in which they will be used, the system performs quite well. However, when the objects are initially distributed to the processes, the performance is not as good as it might otherwise be.

Consider the example above, where an object is the root of a complex scene graph. Imagine that we want to swap the location of two such complex graphs in a graphical scene. Because of the above optimization, the arguments to the group GO `replace` method will only have their global identifiers sent, each of which is 8 bytes long, resulting

in a small update message. Without the above optimization, both of the complex graphs would need to be pickled into the update message, making it much larger. This large message would then need to be copied across the network, and unpickled in the remote processes, only to have these objects thrown away because they already exist in the destination processes.

If we look at the steady state behavior of our shared graphical applications, the time critical activities that occur involve manipulating existing objects, changing their position or appearance, and so on. We have discussed above how the system is optimized for the manipulation of objects that already exist in the destination processes. However, the design is also geared toward efficient manipulation of the properties of objects, as discussed in Section 5.3.1. When we change the behavior of a property (to move an object or change it's appearance, for example) the only data sent over the network is the behavior object, which is an unsynchronized replicated object and is therefore passed relatively quickly (because there is little overhead required to create a new unsynchronized replica).

Unfortunately, when large Repo-3D objects are passed across the network, we have found that the performance is relatively poor. There are a number of reasons for this. First, the objects themselves are quite large because of relatively verbose data structures used to embed the DistAnim-3D objects in Repo (see Figure 5-5). While we optimize the transfer of objects somewhat by passing as little information as possible (recreating it in the destination process), there are times when we cannot get around sending complex Repo objects over the network. This is compounded by the unoptimized message transfer protocols of the Shared Object runtime system, which require all messages to be sent to the sequencer and then forwarded to the clients. If the message is large, reading it from, and writing it to, the network inside the sequencer is costly, especially since we are implementing these activities at the application layer, which requires the data to be copied between memory buffers multiple times.

A third reason for the poor performance is that most of the objects in Repo-3D scene graphs (i.e., GOs, property values, and animation handles) are synchronized, and there is currently no facility for programmers to create unsynchronized versions of these objects, even when they know it is safe to do so. This is a problem because there is a non-

trivial amount of overhead involved in setting up new replicas of synchronized replicated objects, whereas there is little overhead involved in setting up new replicas of unsynchronized replicated objects. As programmers design, and later optimize, their applications, there are often significant portions of the graphical scene that do not change, and therefore do not need to be implemented using synchronized objects.

Consider the Distributed CATHI example in Section 5.4.4. The main animation object is a complex data structure with many GOs and properties embedded within it. When one of the CATHI viewers starts up, it typically takes about 30 seconds for this object to be copied across the network from the viewer that reads it from disk. However, none of these GOs or properties are modified while the viewer is running; the only changes made are to the animation handle, or are local variations. If it was possible to have unsynchronized Repo-3D objects, only the animation handle object would need to be synchronized in this example.

A final performance issue arises when dealing with collaborators who are far apart. Since all of the updates to an object must pass through that object's sequencer, when a process updates an object whose sequencer is located far away, there is likely to be a significant delay associated with that update. Local variations, and clever design, can ameliorate this problem somewhat, but, this problem will exist, in some form, in any system that enforces strict consistency across replicas.

5.7 Discussion

In this chapter, we have discussed the design, implementation and performance of Repo-3D, the distributed, interactive 3D graphics component of Coterie. Since Repo-3D's objects are directly distributed, Repo-3D simplifies rapid prototyping of distributed, interactive 3D graphics applications by circumventing the "dual database" problem and allowing programmers to concentrate on the application functionality of a system, rather than its communication or synchronization components. We have introduced a number of issues that must be considered when building a distributed 3D graphics library, especially concerning efficient and clean support for data distribution and local variations of shared graphical scenes, and discussed how Repo-3D addresses them. We have shown how Repo-

3D builds on the Shared Object package, and how its implementation is a good example of the flexibility and usefulness of the Shared Object package design.

However, while we have found the Repo-3D facilities to be extremely useful, they are not a panacea for all the problems of building distributed graphical applications. While it is possible to ignore the existence of the network during the initial exploratory programming phase, programmers must still be conscious of their design choices if they wish to achieve good network performance. In particular, when using Repo-3D, programmers need to keep the limited bandwidth of the network in mind.

Network bandwidth is an important issue because Repo-3D data structures are often large, so passing them around the network can be costly. When designing standalone applications, complex scene graphs can be added to or removed from a graphical scene with little thought to their size (aside from possible rendering performance implications). This is not true in a distributed application, as distributing new scene graphs to a process can take significant amounts of time. This impacts the design of Repo-3D programs in a number of ways:

- Data should be distributed early. As discussed above, programmers should arrange to distribute large data structures and scene graphs during program initialization, so that they do not need to pay the price during the steady state of program execution.
- Data should be distributed once. If a scene graph that is being removed from the graphical scene may need to be added back later, a programmer should arrange to hold a reference to it in the process, so that it does not get garbage collected.
- Data should be distributed incrementally. If a large data structure needs to be passed around the network, programmers should consider breaking the scene graph up and passing parts around separately. For example, if it is necessary to provide feedback on the progress of copying a large data structure, the data structure must be broken up because each method call made to pass an object is atomic, and there is no way to provide feedback as to its progress. If a complex scene graph is broken down, the program can provide feedback as each piece is transferred. Consider the CATHI animation viewer in Section 5.4.4; the animation used in the example takes approximately 30 sec-

onds to send across the network during program initialization. Many people who saw the system being demonstrated commented that it would be useful if we provided feedback about the progress of the initial transmission of the animation between processes, which we can not do because it is transferred as a single object.

While these design decisions are necessary because we are in a distributed context, they are not substantially different than the kinds of activities programmers engage in when optimizing stand-alone applications, such as initializing data structures at the beginning of program execution, caching objects that are expensive to recompute, and so on. In general, we have found that optimizing Repo-3D programs is similar in flavor to what programmers are used to doing with non-distributed applications, but requires thinking about different kinds of performance issues and bottlenecks than one would think about when optimizing stand-alone applications.

CHAPTER 6 Coterie Examples

"Simple things should be simple, complex things should be possible." – Alan Kay

---

In the previous chapters, we have discussed the Shared Object package, Repo and Repo-3D. Repo and Repo-3D are built on top of the Shared Object package and are two of the key components of Coterie, our system for exploratory programming of distributed augmented environments, introduced in Chapter 2. For each of these components, we have given examples of their use in the corresponding chapter. In this chapter, we will describe two examples of current work in our lab, created with Coterie, that illustrate how these components work together.

6.1 Of Vampire Mirrors and Privacy Lamps

One of the current research projects in the Computer Graphics Lab is called EMMIE (Environment Management for Multi-user Information Environments), and is concerned with the exploration of user-interface issues that arise in collaborative augmented reality systems, such as how one deals with information privacy [Butz et al., 1998]. The prototype is an interesting example of the use of Repo and Repo-3D. We will focus our discussion of EMMIE on one aspect of the system, namely how replicated object directories (the ODs from Section 4.6.5) are used as the basis for information sharing. In the course of that discussion, we will show how Repo-3D is used to construct the main objects for this system.

An EMMIE application is built around a data structure the authors call a VUB[1], which is an OD containing Repo objects with a well defined structure. The objects contained in the VUB are replicated objects representing both the virtual and physical items in

1. The origin of this name is lost, but it has been retained for historical reasons.

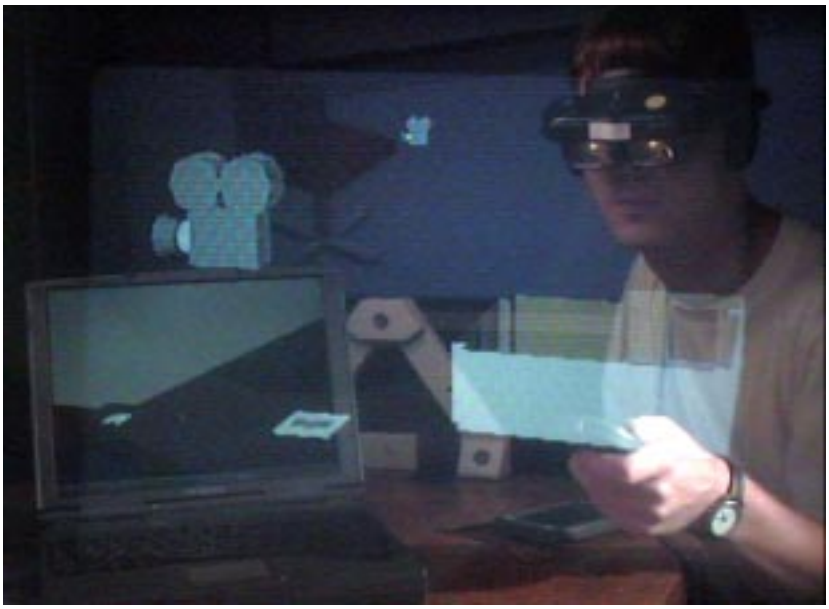


Figure 6-1: The EMMIE system for collaborative augmented environments. Here we see a view of a user of the system, taken from the viewpoint of a second user. Both are wearing see-through head-worn displays, and see a shared augmented environment. In this scene, there are generic icons in the world representing images (the photographic slide) and movies (the video cameras), as well as other objects, such as the model of our lab currently being manipulated by the user. EMMIE integrates this virtual information with other displays. This allows, for example, the information associated with the icons to be viewed on the laptop that is sitting on the desk.

the world. The EMMIE clients can be thought of as viewers that allow people to interact with these objects. An image of the current prototype is shown in Figure 6-1. Each of the virtual items (the model being manipulated by the user, the two small video cameras and the photographic slide) are VUB items. Any process can create a new VUB item and add it to the shared OD, which will cause it to appear in all other viewers.

The routine used to create a VUB item is shown in Figure 6-2. When a process creates an item, some informational attributes are specified (such as a symbolic name, a type, and the owner), in addition to a Repo-3D graphical representation (`localGO`) and an initial 3D position. In addition, arbitrary data can be added to a VUB item using the `raw` field. Since the VUB item is replicated, clients can update these values at any time (in an object or application dependent manner) and all copies will receive the changes.

```
let hilitGO = proc(go)
  let res = GroupGO_New();
  let box = GroupGO_New();
  res.add(go);
  box.add(go);
  res.add(box);

  GO_SetTransform(box, Matrix4_Scale(Matrix4_Id,1.4, 1.4, 1.4));
  SurfaceGO_SetTransmissionCoeff(box, 0.6);
  SurfaceGO_SetEdgeColor(box, "white");
  SurfaceGO_SetEdgeVisibility(box, true);
  res;
 end;

let newItem = proc (name, itype, owner, localGO, pos, raw)
  (* the GO itself is a choice group, 0=visible, 1=hidden *)
  let publicgrp = ChoiceGroupGO_New(0);
  let invgo = GroupGO_New();
  publicgrp.add(localGO);
  publicgrp.add(invgo);

  (* the hilightable GO is a choice group, 0=normal, 1=hilit *)
  let highlightgroup = ChoiceGroupGO_New(0);
  let hilitgrp = hilitGO(publicgrp);
  highlightgroup.add(publicgrp);
  highlightgroup.add(hilitgrp);
  highlightgroup.setName(name);
  GO_SetPickable(highlightgroup,true);
  GO_SetTransform(highlightgroup,
    (Matrix4_Translate(Matrix4_Id, pos[0], pos[1], pos[2])));

  {replicated,
    name => name,
    type => option itype => 0 end,
    owner => owner,
    GO => highlightgroup,
    filename => ok,
    raw => raw}
 end;
```

Figure 6-2: The routine to create a VUB item. In EMMIE, well defined objects are contained in an OD (referred to as a VUB). These items are replicated objects containing a name, a type, an owner, a filename (initially unset) and a field containing arbitrary data (`raw`). The GO for the object is embedded in a hierarchy of GOs that encode the visual representation of the highlit/normal and public/private states of the item using ChoiceGroups.

Before adding the new item to the OD, a new GO scene graph is created to support the needs of the EMMIE system, as described in Figure 6-3. This hierarchy is created once at the site where the item is created, freeing the viewers from having to create the hierarchy for each item they import from the OD. More importantly, the choice groups are repli-

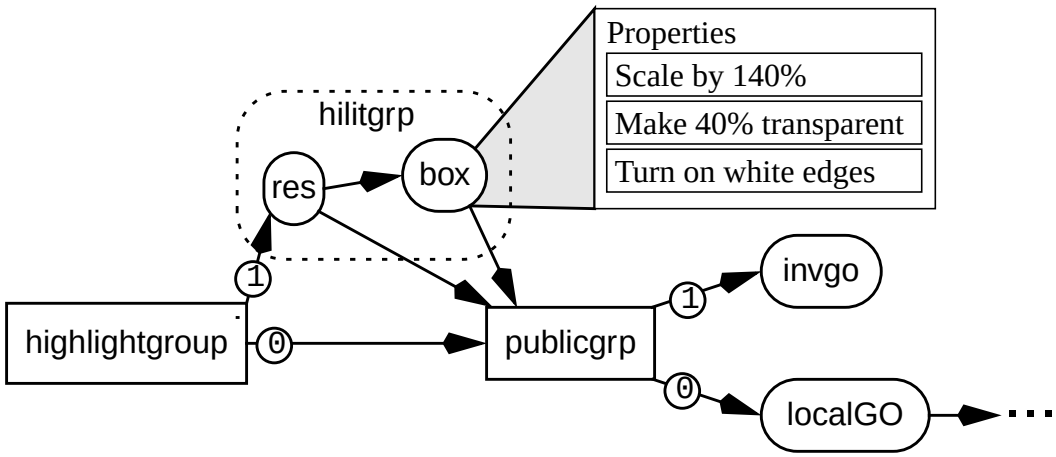


Figure 6-3: The structure of a VUB item's GO. The names in this diagram refer to the variable names in the code that creates this hierarchy (shown in Figure 6-2). `localGO` is the arbitrary hierarchy of GOs representing the item, passed into the `newItem` procedure. `highlightgroup` and `publicgrp` are ChoiceGroups. `publicgrp` chooses between displaying the object (choice 0) or hiding it by displaying the empty group `invgo` (1). `highlightgroup` chooses between displaying the object (0) or displaying the highlighted object (1). Highlighting is accomplished by giving the group `res` two children, the object to be highlighted and another group node, `box`, that creates the highlight. `box` has one child, the object to be highlighted, and has properties that scale it, make it transparent and turn on white polygon edges. Thus, no matter what the child looks like, or how it changes, `box` will be a enlarged ghost with white edges around it.

cated with the objects, which allows one of them (the `publicgo` choice group) to be used by the clients to experiment with privacy techniques. The two choice groups are used as follows:

- `highlightgroup` is used for highlighting objects as the user interacts with them. Each user has a 3D selection device that they use to manipulate objects in the virtual world. For example, the user in Figure 6-1 is manipulating the room model with their 3D device. The global `ChoiceGroupGO_Display` property of this node is set to 0, selecting the `publicgrp` node for display. As they move the pointer through the scene, objects are highlighted by locally setting the `ChoiceGroupGO_Display` property to 1, selecting `hilitgrp` instead.
- `publicgrp` is used for hiding and revealing the objects. In this implementation of EMMIE, an item is visible everywhere unless it is somehow made private by one of the

users (the various ways of managing privacy is the topic of [Butz et al., 1998]). The global `ChoiceGroupGO_Display` property of this node is normally set to 0, selecting the `localGO` node for display. When a user makes an item private, they set the global `ChoiceGroupGO_Display` property of `publicgrp` to 1, selecting the empty `invgo` group node for display everywhere, and set the local value of the `ChoiceGroupGO_Display` property to 0, making it visible locally.

As can be seen, fairly complex and interesting interaction behaviors have been implemented by combining a relatively simple object hierarchy with a judicious use of local and global property values. Also, notice that while these techniques are well suited for experimentation with these interaction issues, they are not well suited for creating a final, deployable system, as these techniques reply on cooperation between the various processes and provide no security. For example, there is nothing to stop one of the clients from changing the GO hierarchies arbitrarily, breaking the system. Or, more subtly, one of the clients could set their local `ChoiceGroupGO_Display` property on the `publicgrp` node to 0, thus ignoring the global setting and always making the object visible. However, these issues do not concern us at this stage of interface design, as we are interested in building and evaluating prototypes to explore different interaction techniques.

The example also demonstrates how simple, well defined objects can be used with the ODs to create a relatively powerful application. By combining the ODs with Repo-3Ds graphical objects, the graphical components of the application are straightforward to create, and the programmers can focus on other tasks, such as how to specify privacy, how to integrate other displays such as laptop computers into the environment, and so forth.

6.2 Shared Sketch

One of the tasks that we would like to perform in our augmented environment prototypes is informal collaborative creation of 3D objects. Unfortunately, intuitive and powerful interfaces for this task are non-trivial to implement. One example of an interface for informal creation of 3D objects, using a sketching metaphor, is the Brown University Sketch system [Zeleznik et al., 1996]. Rather than develop a new system (which would have been

Object	Specification
Cube	Parameters: none The canonical cube between [-1,-1, -1] to [1, 1, 1]
Cylinder	Parameters: none The canonical cylinder with base at [0,-1, 0], tip at [0, 1, 0] and radius of the base of 1
Cone	Parameters: none The canonical cone with bottom at [0,-1, 0], top at [0, 1, 0] and radius of 1
Truncated Cone	Parameters: `rad` The cone is centered around the origin, with its bottom face a circle of radius 1 centered at (0,-1,0). The top face of the truncated cone will always be in the plane Y=1, centered on the Y axis, and will have a radius of `rad`.
Truncated Pyramid	Parameters: `taperX taperZ shearX shearZ` The pyramid is centered around the origin, with its bottom face a square going from [-1,-1,-1] to [1,-1,1]. The top face of the pyramid can be of any size, but will always lie in the plane Y=1. The taper vector determines how much the top face of the pyramid should be tapered in from the default of [1,1]. The shear vector determines how much the top face of the pyramid should be offset from the Y axis.
Extrusion	Parameters: `num_pts x1 z1 x2 z2 x3 z3 ...` The profile lies in the plane Y=-1. The length of the extrusion should always be 2 units, up the Y axis to the plane Y=1.
Surface of Revolution	Parameters: `num_pts x1 z1 x2 z2 x3 z3 ...` The profile lies in the half plane X>0, Z=0. The axis of revolution is the Y axis, and the profile is swept 360 degrees

Table 6-1: Sketch Object Definitions. Sketch supports seven objects. All Sketch scenes are created using these basic objects, and Boolean set operations between them.

a prohibitive amount of work), we worked with members of the Brown Computer Graphics Lab to integrate the Sketch system into our environment.

The Sketch system allows the user to sketch 3D scenes using gestures. The system allows the user to create seven basic types of objects, detailed in Table 6-1. More complex objects are created by performing the Boolean set operations of constructive solid geometry (CSG) operations on instances of these seven object types, such as taking the union of two objects or subtracting one object from another object. All operations, including object

creation, deletion and CSGs, are done using gestures. The system infers object grouping based on how and where the objects are created and allows interactive and intuitive specification of CSG.

To integrate Sketch with Coterie, we defined a TCP protocol that describes all of the logical operations Sketch can perform, such as creating, deleting, hiding or showing objects, changing their position or color, changing object grouping, performing CSG operations, and so on. Loring Holden (a researcher at Brown) implemented this protocol inside the Sketch application, and we implemented it in Repo. The Repo module we developed uses a Repo-3D GO for each object in Sketch, and a Repo-3D group GO to hold all of the objects in a Sketch scene. The module is symmetric, ensuring that the contents of that group match the contents of the Sketch world, regardless of which side changes the scene. When the system is started, it obtains the current scene from Sketch and creates an initial set of Repo-3D objects corresponding to the Sketch objects. When the Sketch user modifies the scene in any way, the corresponding changes are made to the Repo-3D objects. The module watches the Repo-3D objects for a well defined set of changes (of the sort that the module itself would perform) and issues commands to Sketch to reflect these changes.

From a Coterie programmers point of view, this module maintains a group GO corresponding to a Sketch scene. This group GO contains all the information needed to recreated the scene in a running instance of the Sketch program. Therefore, after the module is initialized and connected to a running Sketch, a group GO containing a valid Sketch scene can be passed to the module, and the associated Sketch will have its current objects removed and the new objects (corresponding to the group GO) loaded into it. New scenes can be created by passing an empty group GO to the module. Therefore, we can maintain many different Sketch scenes in Coterie and switch between as needed.

Since the group GOs representing Sketch scenes are Repo-3D object hierarchies, they may be passed to any Coterie process and used like any other Repo-3D objects. Currently, a group GO representing a Sketch scene should be treated as an immutable object to ensure that it remains a valid Sketch scene, since the objects in a Sketch scene are defined in relation to one another (including their grouping and CSG specification). However, the group GO can still be added to other Repo-3D scenes, watched for changes, and

so on. It may also have other non-Sketch objects added to it, as these will be ignored by the Sketch synchronization module.

As described so far, this is an interesting example of software engineering and "program reuse". We have created a module within Coterie that is a black box and appears to *be* the Sketch system, from the viewpoint of the Coterie programmer. However, if we look at how the Sketch objects are implemented in Coterie, we will see that this is also an interesting example of how Repo-3D's distributable and extensible objects are used in practice.

Since the group GOs containing Coterie Sketch scenes can be passed around the network, and will continuously reflect the changes made by the Sketch user, we can see how the distributability of Repo-3D objects is useful. However, the mere fact that the objects are distributable is not sufficient to allow them to be used to represent the Sketch scenes. As mentioned above, a Sketch scene is composed of a set of the seven basic objects listed in Table 6-1, plus CSG operations on them. Furthermore, objects can be grouped together, used to interactively perform CSG operations on each other (referred to as the object being a "cutter"), and be visible or hidden. Therefore, if we want to fully represent the Sketch scene with a Repo-3D group, this additional information needs to be embedded in the Repo-3D objects.

To carry this information with the Sketch scene, each Repo-3D object representing a Sketch object is extended with a field named `info` that contains a replicated object holding this additional state, as shown in Figure 6-4. Every Sketch object is represented as a choice group GO with two children, one for the Repo-3D object corresponding to the Sketch object, and one to hold the CSG result if this object is used in a CSG operation. The objects created in Figure 6-4 contain sufficient information to recreate the Sketch scene. Furthermore, since the additional information is contained in replicated objects, when any site changes a Sketch object, all sites will be notified of the change. Furthermore, if multiple processes are editing the same Sketch scene, changes made by one user will be forwarded to the other users' Coterie processes, and from there into their Sketch programs. Thus, each user will see changes made by other users in real time.

```
let obj = ChoiceGroupGO_New(-1).extend(
     {simple, info => {replicated,
        name => name, (* our sketch object name *)
        type => typ, (* our sketch object type *)
        group => simple[], (* which objects we are grouped with *)
        cutter => false, (* are we a cutter? *)
        cutting => ok, (* the name of the object we are cutting *)
        data => objData, (* object data *)
        visible => false, (* are we visible? *)
        csgOp => ok, (* are we CSGed? *)
        csgGeom => ok (* the geometry of our CSG result *)
      }
     });

let t = TransformProp_NewConst(Matrix4_Id);
obj.setProp(GO_Transform, t);

let c = ColorProp_NewConst("white");
obj.setProp(SurfaceGO_Color, c);
obj.setProp(SurfaceGO_BackColor, c);

let objGO = GroupGO_New();
let csgObjGO = GroupGO_New();
obj.setName(name);
objGO.setName("go");
csgObjGO.setName("csg");
obj.add(data.objGO);
obj.add(data.csgObjGO);
```

Figure 6-4: The definition of a Coterie Sketch object. The Sketch object is represented as a Repo-3D choice group GO, where the two possible children of the group are the object itself, and a CSG result object. The object is extended with a replicated object containing the information needed to reconstruct the object in Sketch. A transformation property and color property are assigned to the object, and the property value is set to the new value when any instance of Sketch changes it.

The extensibility of Repo-3D objects is also used to create four of the seven basic Sketch objects. If we look at the list of Sketch objects in Table 6-1 and compare them to the Repo-3D objects in Appendix H.1, we see that only the first three (cones, cylinders and cubes) are available in Repo-3D. We created a new module for each of the other four, using group and indexed polygon set GOs (see Appendix H.1.12 for a description of indexed polygon sets).

The implementation of one of these new objects, the Truncated Pyramid, is described in Section 5.4.3. A truncated pyramid is created as an indexed polygon set, and extended with a new `data` field holding a replicated object that contains the parameters

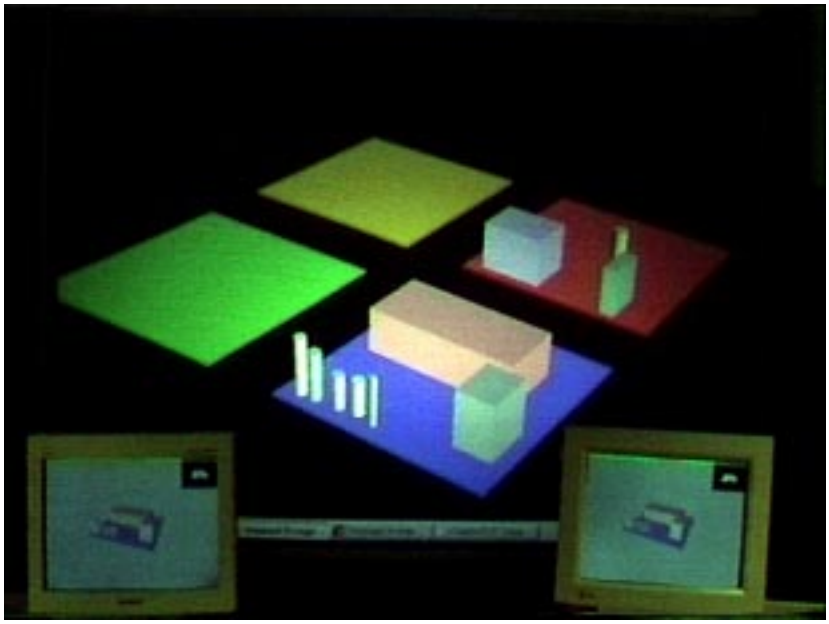


Figure 6-5: Distributed Sketch in use. The wall-sized display shows the shared Sketch world, containing four sketchpads. In this image, the two client workstations in the foreground are editing the same sketchpad (the one in the foreground of the shared world), but they could also be editing different ones. The system supports any number of simultaneous clients editing the sketchpads in any combination.

defining the truncated pyramid. The vertices of the indexed polygon set representing the pyramid are defined using asynchronous properties that compute their values based on the values of the items in the `data` field. Therefore, when the values in the `data` object are changed, the pyramid changes shape accordingly. And, since the `data` field is replicated, if we pass the object around the network, the values can be changed at any site and all sites will be updated (and have the shape of their pyramid changed accordingly). The other three objects are defined similarly, with the parameters that define the objects contained within them.

As we described above, the goal of the module is to allow the Brown Sketch system to be used to create 3D models that can be easily integrated with our research prototypes. To demonstrate the ease with which this can be done, we created a simple demonstration program that allows any number of users to cooperatively edit a group of four Sketch scenes, represented as colored “sketchpads”. An image of this demonstration

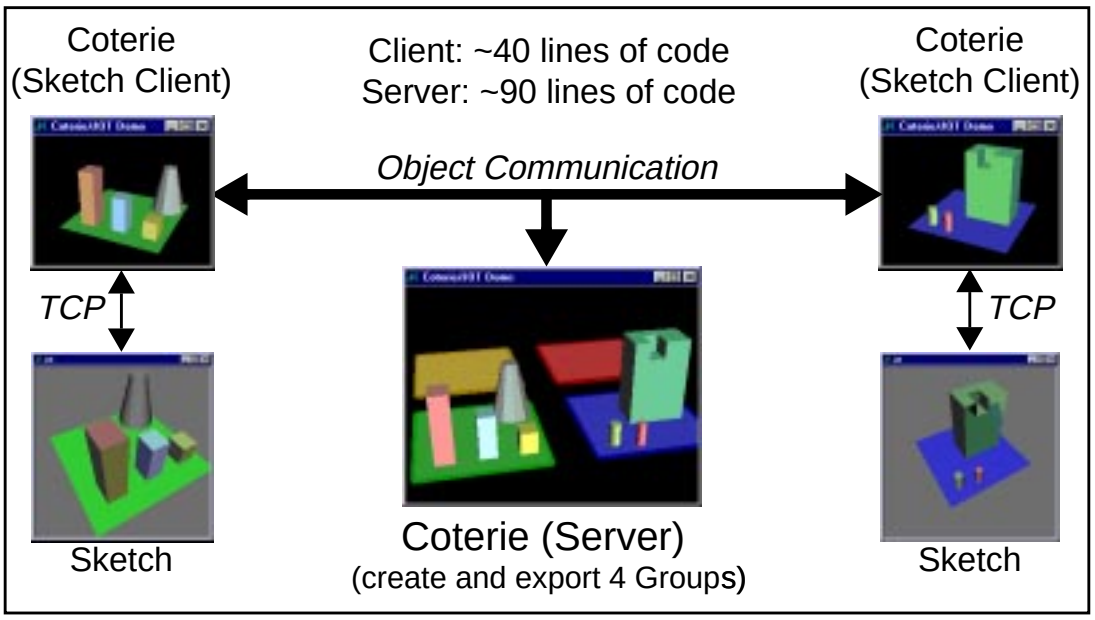


Figure 6-6: The structure of the Distributed Sketch prototype. The organization of the Distributed Sketch prototype shown in Figure 6-5. The server creates four sketchpads as group GOs and exports them to the network. Each client is composed of a Coterie program and a Sketch program, connected together by a symmetric TCP protocol that keeps their sketchpad synchronized. Therefore, if the client switches sketchpads, by importing a different one, the sketchpad in the Sketch process is loaded with the new scene. Conversely, if the user modifies the scene in Sketch, the sketchpad in the Coterie process is updated. In this case, all other replicas of the sketchpad are also changed (such as the one in the server, or in any client that is also editing the same sketchpad), because the sketchpad objects in Coterie are built using Repo-3D objects.

program is shown in Figure 6-5, where a wall-sized projection screen is displaying the four Sketch scenes on their sketchpads, and two clients are running on the smaller displays in the lower corners of the image. In this image, the two clients are editing the same scene in their Sketch programs (the one on the sketchpad in the foreground of the large, rear image). Each client screen contains a large Sketch window, and a small Repo-3D window (in the upper right corner of the display). This small Repo-3D window is used to select which of the four sketchpads should be edited in the local Sketch program.

The structure of this application is shown in Figure 6-6. There are two simple programs in this prototype, a server (implemented in about 90 lines of Repo code), and a client program (implemented in about 40 lines of Repo code). The server code creates four

sketchpads, exports them to the network and creates the display shown. The client creates a small window and waits for input telling it which of the sketchpads to import from the network and pass to the Coterie Sketch module (causing it to be loaded into Sketch). This trivial prototype took very little time to write, and allows any number of clients to edit the shared sketchpads, including have more than one editing the same sketchpad simultaneously. Any changes made to a sketchpad are reflected on the large server display, and in any clients sharing that sketchpad, in real time.

What this prototype does not implement is any form of floor control. While Coterie guarantees that the objects will remain synchronized, it is up to the programmer to implement whatever kind of shared editing policies they desire. For example, the replicated mutexes of Section 4.6.4 can be used to allow one site to lock a sketchpad, or objects within the sketchpad, when they want to edit them. The important thing to remember here is that Coterie supports the programmer in creating whatever policies they desire, but does not impose any on them.

CHAPTER 7 Conclusions and Future Work

In this dissertation, we have examined the design and implementation of various components of Coterie, our research platform for building prototypes to explore the user-interaction issues of multi-user augmented environments (AEs). We based our design on our previous experience building single-user augmented reality (AR) prototypes, some examples of which were discussed in Section 2.1.

Based on this previous experience, it was obvious to us early on that exploring multi-user AEs would be unusually challenging. On one hand, the physical environments themselves are extremely difficult to work with. Multiple users, multiple displays of different kinds (from see-through head-worn to wall-mounted to hand-held), and a wide variety of input devices (from pens and mice to voice to three and six degree-of-freedom sensors) must be integrated into a single cohesive system. On the other hand, these devices and displays are attached to an assortment of computers, requiring that even the simplest of applications be distributed over many machines. It is this latter problem that most concerns us, as building distributed applications can be extremely difficult, especially in highly interactive application domains such as ours. This difficulty is exacerbated by the exploratory nature of prototyping systems to investigate a completely new interaction paradigm: neither the structure of the applications, the kind of data being shared, nor the distribution characteristics of that data are necessarily known ahead of time, and will likely be modified continuously as the applications are developed.

Our solution to this problem is embodied in our Coterie testbed. The fundamental design choice we made with Coterie was to first create a general purpose, easy-to-use, flexible and efficient programming environment as the lowest level of our system, and then build the other tools we need on top of it. To ensure flexibility and ease of use, we built this lowest layer by tightly integrating transparent support for data distribution into the

object system of Modula-3. We elected to integrate data distribution with a popular programming language so that we can take advantage of both existing software and programming skills.

Transparent integration is only possible if the programming language supports a programming model that is suitable for distributed programming. Modula-3's programming model—multiple threads of control communicating via shared objects—is ideal. By providing an object-based implementation of distributed shared memory (DSM), often called a distributed object memory (DOM), both stand-alone and distributed programs are built the same way, with local and distributed data being used transparently and interchangeably, and with threads on the same or different machines communicating through shared objects.

The DOM approach is not revolutionary in and of itself: over the past two decades, many others have recognized the importance of integrating data distribution into the programming languages they use. Starting with the popularity of the Remote Procedure Call system (RPC) [Birrell and Nelson, 1984], client-server data sharing packages have been built for many programming languages, and have become more tightly integrated and easier to use along the way. Packages for modern, multi-threaded languages, such as Java RMI [Wollrath et al., 1996] or Modula-3 Network Objects [Birrell et al., 1993] are almost transparent to the programmers using them.

Unfortunately, these popular packages are not sufficient for our needs because we require both client-server and replicated data, and none of the DOM packages created for popular languages support replicated data. This is understandable, as replicated data sharing is significantly harder to implement than client-server data sharing. All client-server data sharing packages use the same general approach: a data item exists in one process, and any access to that data in any other process is forwarded to the process containing the data. Support for replicated data is not as simple, as a variety of design trade-offs must be made that do not arise with client-server data, affecting the latency of update distribution, efficiency of access and network usage, data consistency, fault tolerance, and so on.

The first contribution of this dissertation, therefore, is the creation of the Shared Objects DOM package for Modula-3. The Shared Objects package supports replicated data, and the design trade-offs mentioned above have been made with the needs of our application domain in mind. In particular, the Shared Objects package focuses on providing a high degree of flexibility to support exploratory programming, as well as providing low latency update distribution and strictly consistent data. When combined with the existing Network Objects client-server package, we have the solid foundation we need for distributed programming: a DOM programming system supporting client-server, unsynchronized replicated and synchronized replicated data.

On this foundation, we built the other major components of Coterie, the most important of which are Repo and Repo-3D. Repo is an interpreted language supporting the same DOM programming model provided by the Shared and Network Objects packages. Repo-3D is a novel, high-level 3D graphics package in which all the graphical objects are both extensible and directly distributable (since they are implemented with Shared Objects). This allows graphical application programmers to encode application state in the 3D graphics objects and use these objects directly as part of the distributed data structures of their application.

The combination of Repo and Repo-3D allow distributed, interactive graphical applications to be built with a minimum of effort because programmers do not need to overly concern themselves with issues of data distribution, and can therefore expend the vast majority of their programming and design efforts on application development. Since more time and effort can be expended on the applications themselves, rather than the mechanics of data distribution, previously impractical applications become possible.

Repo and Repo-3D are both contributions for two reasons: each is an new and interesting research result in and of itself, and both are examples of how a flexible and easy to use infrastructure, combined with an existing programming language, enables previously difficult problems to be tackled in a straightforward manner. Both Repo and Repo-3D are based on existing Modula-3 packages (Obliq and Obliq-3D, respectively), and modifying them to support replicated data was feasible because of the flexibility of the Shared Object package, and its tight integration with Modula-3.

If we had chosen to build a new language, or had added support for replicated data to Modula-3 in a way that was inflexible or not compatible with the predominant programming style, creating these packages would have been significantly more difficult. Given the usefulness of Repo and Repo-3D, we hypothesize that the main reason no similar packages have been built previously is the lack of data replication facilities such as those provided by the Shared Objects package; building them is simply too hard without such facilities, because the details of managing replicated data are too complex. As a result, the people who would benefit most from these tools (researchers and developers of distributed interactive applications such as ourselves) do not embark upon building them, but rather focus on building custom solutions to solve their immediate problems.

The flexibility of the Shared Objects package derives primarily from the fact that consistency is defined in terms of method execution (both the order of execution and whether the method modifies the global state), with almost nothing being said about the contents of an object's data fields. For example, the programmer of an object has great flexibility in partitioning the work into parts executed once (at the calling site) and parts executed at all sites, by taking advantage of the fact that update methods are broadcast and executed at all sites while read methods are not. A read method can therefore perform some work locally, and then call an update method to perform the rest of the work globally. This same technique can be used to lessen the impact of the restrictions on update method argument types; for example, a read method can manipulate the restricted argument locally and use the results as arguments to an update method.

We make use of the ability to perform arbitrary actions in methods in the implementation of both Repo and Repo-3D, but especially in Repo-3D (as discussed in Section 5.5). Since part of the state of each graphical object is global, and part is local to each machine (both the part that associates the conceptual graphical object state with the concrete state used by the rendering subsystem, and the local variations to the global state), we can manage these data structures in a straightforward and efficient manner by manipulating local data within the read methods and global data within the update methods.

Throughout this dissertation, we have demonstrated the simplicity and flexibility of the various components of Coterie through illustrative examples. These include simple examples that demonstrate important techniques, such as the recurring example of tracker report distribution, the distributed mutex examples of Section 4.6.4 and Appendix F, and the complete single and multi-user prototypes of Section 2.6 and Chapter 7 that members of our research group have built using Coterie over the past few years. We have found that having a system in which distributed and stand-alone applications can be built using a common high-level programming model has greatly simplified development, and allowed us to explore applications and domains that would otherwise have been intractable. Perhaps more importantly, programmers with varying levels of experience, especially those with little distributed programming experience, have used the system successfully. Our programmers have included undergraduate, masters, doctoral and post-doctoral students.

7.1 Future Work

In this dissertation, we have developed a system that is well suited to exploratory programming of tightly-coupled, distributed, highly interactive systems. Our choice of the distributed object memory (DOM) programming model, and the approach we took to providing replicated data within that model, were guided by both the application domain and the exploratory style of programming in which we engage. In the future, we hope both to continue building on this approach to prototyping distributed interactive applications, and to explore different programming models that may be more appropriate to different domains and programming styles.

This latter question is an important one. While tightly-coupled, strictly consistent objects that are distributed using a DOM programming style are useful for exploratory programming, they may not be the most appropriate choice for other domains. For example, if one is building long lived, production quality systems, the trade-offs made between ease-of-use and efficiency might be different; efficiency of execution and network utilization are likely to be much more important than the ease of changing objects from one distribution semantic to another, not to mention the increased importance of other issues such

as fault tolerance. Therefore, the transparency with which the objects are integrated into the programming languages may not be the most important issue, as it is for us.

However, returning to the programming style with which we are familiar, there are a number of ways we envision improving our implementation of the DOM programming model: by decreasing the latency of update distribution, improving network awareness, adding additional per-object replication semantics, extending the programming model to support multi-object operations, improving the flexibility of the consistency guarantees, and improving the handling of time. Finally, we would like to explore these ideas in other programming languages, especially Java.

7.1.1 Shared Object Update Latency

When designing the Shared Objects package, we were extremely concerned that update distribution might be too slow because of the requirement that all updates travel over the network at least twice, passing through one or more sequencers on the way. As it turned out, this has never been a significant problem with the applications we have developed, so we have not needed to address it.

However, we did design one possible solution into the runtime, which has been partially implemented. Our solution is to allow a programmer to designate a replica of a Shared Object as requiring updates in a *timely* fashion. By either designating a replica as the primary updater, or by having the runtime notice that one site is performing most of the updates, the system would be able to arrange for update events to be sent directly from the primary updater to those replicas requesting timely updates. By having the primary updater handle the sequencing for this object, we would bypass the sequencer and decrease the typical network hops from two to one. (Updates by any process other than the primary updater will now take longer, having their network hops increased from two to four because the sequencer must now route update events through the primary updater.)

This facility is only needed in cases where minimizing lag is critical. For example, it may be used when a head tracker is connected to a different machine than the graphics display. In this case, only the primary updater will update the object, so the increased num-

ber of network hops for other updaters is not an issue. This facility was not implemented because Moore's Law obviated the need for it: between the time we designed the system, and the time it would have been implemented, the machines being used became sufficiently powerful that it was always the case that a head tracker could be attached to the same machine (and thus read from the same process) that generated graphics for that user's display.

An alternative approach, which would obviate the need for the above facility, would be to support per-object sequencer migration. In the general case, as an application evolves over time, it is possible that the sequencer for an object may no longer be located in a cluster that contains processes issuing updates on that object. For both performance and network utilization, it would be best if the sequencer for an object is located in the cluster that contains the processes that are issuing most of the updates on that object. Therefore, the system should be able to migrate the sequencing duties for an object to the sequencer for the cluster where the updates are being performed. Given such a facility, and the fact that all processes are capable of sequencing updates, it would be a small step to notice that one particular process is issuing most, or all, of the updates for an object, and allow it to do the sequencing for that object. Such techniques are in many ways similar to the optimistic locking of objects done in many CSCW and distributed systems, where an object is allowed to be updated only if the process holds the lock on the object, and the system arranges to acquire the lock when the process attempts to update the object. No sequencer-based systems that we know of allow the sequencing duties to migrate into a client process in this manner.

7.1.2 Network Awareness

Another area that we would like to address is that of *network awareness*, or the amount of information a programmer can obtain about the network behavior of the program. Currently, the Shared Object package provides a basic level of network awareness, following the approach of the Network Object system: when a distribution problem is detected, the runtime raises a `SharedObj.Error` exception, analogous to the `NetObj.Error` exception raised by the Network Object package. This allows a programmer to react to

problems, but does not require using a radically different programming model than they are used to. The Shared Object package further exposes the network to the programmer by supporting the definition of custom pickling routines, allowing a programmer to perform (arbitrary) special actions when an object is passed between processes.

One facility that would enhance network awareness, and that we have found a need for, is to allow the programmer to specify cleanup code to be run in a process when an object is removed from that process: if a programmer wishes to do arbitrary things in the pickling routines when a new replica is created in a process, it is sometimes necessary to be able to undo some of these actions when the object is removed from the process. Currently, this is not a major drawback because the prototypes being developed tend not to be long-lived and situations where this is absolutely necessary are rare.

Network awareness is more of an issue when designing objects that need to deal directly with the fact that they are replicated. Unlike Network Objects, Shared Objects exist simultaneously in multiple processes, and it occasionally turns out to be useful to know where these replicas reside, and to be notified when additional replicas are created or removed. Furthermore, we have found that it is also sometimes useful to know if an update was initiated locally or remotely, as well as which remote process initiated the update. This information could be made available to one or both of the Shared or Callback Object methods. These facilities can be useful in implementing permissions and capabilities inside objects, for example, allowing them to present different information to different clients.

Recall the discussion of a distributed mutex from Section 4.6.4. In that section, we describe how to create a fair mutex that will not give preferential access to sites closer to the sequencer. To do this, we need to enqueue requests for the mutex, rather than have sites try to reacquire the mutex when they notice it has been released. But, if we enqueue requests for locking the mutex, we then require notification if a site crashes so that we can remove that site’s outstanding requests from the queue in the replicas at all other sites.

7.1.3 Additional Replication Semantics

Currently, a Shared Object is created by inheriting from the `SharedObj.T` type and following a few simple rules. There are two alternatives for supporting additional semantics. On one hand, procedures could be provided in the Shared Object package, or methods added to the `SharedObj.T` type, that allow the programmer to control the replication semantics of a generic "replicated" object type. Alternatively, new semantics could be provided by creating subtypes of `SharedObj.T` and having the programmer inherit from them as appropriate. We prefer this latter approach, as it is cleaner and more in line with our goal of tight integration with the type system of Modula-3.

Ideally, we would like to extend the replication semantics to be as flexible as the Penumbra system is for client-server objects, allowing programmers to define their own consistency semantics [Kristensen and Low, 1995]. However, while this may be difficult to accomplish while keeping the system easy to use for novice programmers, a number of specific new replication semantics could be provided without supporting programmer-defined consistency. We are particularly interested in replication semantics that fit with the current write-update scheme. Currently, the Shared Objects runtime assumes all updates must be applied to all replicas, and ensures that all updates are applied to all replicas in the same order. During our work, we have discovered two additional semantics that would be particularly useful for distributed interactive applications, which we will call *any-order update* and *latest-only update*.

An *any-order update* scheme asserts that the object in question will remain consistent regardless of the order the updates are applied, as long as all updates are applied. The most obvious example of such an object is an up-down counter that supports increment and decrement operations: as long as all operations are executed, all replicas of the object will be consistent. independent of the order of execution of the operations.

A *latest-only update* scheme asserts that each update operation completely specifies the state of the object, and that only the most recent update is of interest. Such a scheme is useful for rapidly changing objects that satisfy these properties, as missed messages can be thrown away, instead of retransmitted. Furthermore, incoming updates

need never be queued up, since an update can be executed immediately if it is newer than the current state of the replica, and thrown away if it is not. Such an update scheme is also very well suited for efficient implementation using UDP or multicast UDP. Examples of applicable objects include the TrackerPosition object used as an example in Section 3.4.1, and Repo-3D property values. The TrackerPosition object has a `set()` method that completely updates the state of the object, and a `get()` method that retrieves the current position of a tracker. The Repo-3D property values are similar: the current behavior can be set, used or retrieved, but is independent of any other behaviors that might be assigned to the property value at any other time.

The addition of new replication semantics to the Shared Objects package will also affect the other layers of Coterie. Most importantly, such semantics would need to be exposed into Repo to be truly useful to the programmer. Fortunately, adding new semantics to Repo, such as the two described above, would only require that new keywords be added to the language to allow programmers to create (and convert between) objects with the new replication semantics. Currently, Repo uses the `simple` and `replicated` keywords to modify object, array and variable creation, and to convert between distribution semantics. New keywords, and thus new semantics, could be added in a similar manner. For example, `latest` and `anyorder` keywords that could be used in the same way as `simple` and `replicated`, with no other changes being required. These new semantics would be particularly useful for arrays, as many uses of arrays in our programs access and update array elements independently.

Additional replication semantics would also improve Repo-3D. For example, if it were possible to specify the *latest-only update* semantics, the efficiency of the distribution of property values would improve significantly; in this case, updates could be applied (or discarded) when they arrive, without waiting for all previous updates to be applied, and could be applied locally without waiting for the round trip to the sequencer. While programmers may occasionally want all updates to a property value to be applied at all sites (for example, if the changes to the value are being recorded via callbacks), typically only the latest value of a property is of interest.

7.1.4 Multi-object Consistency

There are times when it would be useful to support some sort of consistency or synchronization guarantee across multiple objects. One end of the spectrum we would be to support *causal ordering* (as provided by systems such as Isis [Birman, 1993]), so that we could ensure that multiple updates to distinct objects would appear to happen in the same order in distributed processes if they were causally related. However, the causal ordering algorithms with which we are familiar require full replication, and do not scale well, so it is not clear how one would provide efficient causal order in a system such as ours.

Another option would be to provide a more well defined, restricted multi-object consistency guarantee by allowing the programmer to explicitly specify the group of actions that are to be applied as a unit. Based on our current experiences, it seems reasonable to provide this facility by allowing a thread to mark the beginning and end of a group of actions that should be associated in this way. Since we do not want to modify the language, we would have to use procedure calls to implement this in Modula-3. However, we could easily modify Repo to add a structured statement to support this model, to make the process clearer to the programmer. It may also be useful to implement a transaction model, so that large groups of changes could be applied atomically, or not at all.

7.1.5 More Flexible Consistency Guarantees

Another issue we have encountered with our use of the Shared Object package is that of strict consistency. While we have found the model useful, the local variations we implemented in Repo-3D point out the need for local variations to be directly supported by the object system. Similarly, while many of the data structures we build in Repo benefit strongly from the guarantees provided by strict consistency, we often find ourselves needing to support local variations to replicated objects. Instead of implementing such facilities on an object by object basis, it would be useful if the object system provided these facilities directly. This would greatly benefit exploratory programming, as more applications would then be able to encode their state using the distributed objects, instead of requiring a combination of distributed and local objects.

Another problem we would like to address relates to the transparent use of the Modula-3 type system to enforce consistency. Since we create replicas by passing objects between sites, it is impossible to create replicated objects without creating the object at one site and passing it to the others. As we have found in some of our more complex applications, large objects (such as Repo-3D scene graphs) take a significant amount of time to pass between processes. This is particularly annoying when these objects are statically defined on disk, and the only reason we pass them between processes is to tie the replicas in these processes together.

This is important for both efficiency and software engineering reasons: it is tedious and time consuming (both during program development and execution) to arrange for these replicas to be downloaded from other processes. What we need is a facility to be able to name an object, effectively saying “object A is the same as object B” without having to pass it across the network. Such a facility would allow local object caches to be stored on disk and reloaded on demand, objects to be created from local program files, and so on.

7.1.6 Better Handling of Time

In the current implementation of Coterie, we assume that all of the machines have their clocks synchronized using a time-synchronization protocol such as NTP (the Network Time Protocol [Mills, 1992]). The library uses an internal animation time offset[1] (instead of the system-specific time offset) because different OSs (e.g., NT and UNIX) start counting time at different dates. Unfortunately, this assumption is not always reasonable, especially when mobile computers are involved. We have found that even in our controlled lab environment, the clocks on our machines do not always remain synchronized. This problem is most apparent when using time-based animations in Repo-3D.

To address this problem, hooks have been provided in the `Anim3D` module (see Appendix H.7.1) to allow a programmer to specify their own function to compute the “current” animation time offset within a process. Using this facility, it is possible to build inter-process time synchronization protocols; we have implemented a version of the Sim-

1. Computed as an offset from January 1, 1997.

ple Network Time Protocol (SNTP) [Mills, 1996] using approximately a hundred lines of Repo code (shown in Appendix I). Future systems should integrate more advanced solutions, such as adjusting time values as they travel between machines, so that users of computers with unsynchronized clocks can collaborate.[2] This will become more important as mobile computers increase in popularity, as it may not be practical to keep their clocks synchronized.

7.1.7 Generalized Local Variations in Repo-3D

Another way the current implementation could be improved is in the specification of local variations, which could benefit from adopting the notion of *paths* (as used in Java 3D and Inventor, for example). A path is an array of objects leading from the root of the graph to an object; when an object occurs in multiple places in one or more scene graphs, paths allow these instances to be differentiated. By specifying local variations using paths, nodes in the shared scene graphs could have variations *within* a process as well as *between* processes.

One other limitation of Repo-3D, arising from our use of the Replicated Object package, is that there is no way to be notified when local variations are applied to an object. Recall that the methods of an automatically generated Notification Object correspond to the update methods of the corresponding Replicated Object. Since the methods that manipulate the local variations are non-update methods (i.e., they do not modify the replicated state), there are no corresponding methods for them in the Notification Objects. Of course, it would be relatively straightforward to modify the Replicated Object package to support this, but we have not yet found a need for these notifiers.

7.1.8 Application to Other Languages

While Modula-3 was a popular programming language when this work started, it has declined in use over the years. If we want to popularize these techniques, we need to

---

2. Implementation details of the combination of Network and Shared Objects made it difficult for us to adopt a more advanced solution.

implement them in a more widely used language, such as Java. Java shares many of the advantages of Modula-3 (e.g., threads and garbage collection are common across all architectures) and the packages needed to create a Coterie-like platform are beginning to appear.

While Java does not yet have a replicated object system as powerful as the Replicated Object package, a package such as JSDT [Sun Microsystems, Inc., 1998] (which focuses more on data communication than high-level object semantics) may provide a good starting point. Work is also being done on interpreted, distributed programming languages on top of Java (e.g., Ambit [Cardelli and Gordon, 1998]). Finally, Java 3D is powerful enough to serve as the basis for a library such as Anim-3D, even though its design leans toward efficiency instead of generality when there are trade-offs to be made. For example, the designers chose to forgo Anim-3D's general property inheritance mechanism because it imposes computational overhead. By combining packages such as Java 3D, JSDT, and Ambit, it should be possible to build a prototyping testbed such as Coterie in Java.

References

Arnold, K. and Gosling, J. (1998). *The Java Programming Language*. Addison Wesley, Reading, MA, USA, second edition.

Bal, H., Kaashoek, M., and Tanenbaum, A. (1992). Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205.

Bal, H. E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Ruhl, T., and Kaashoek, M. F. (1998). Performance evaluation of the orca shared object system. *ACM Transactions on Computer Systems*, 16(1):1–40.

Bal, H. E. and Tanenbaum, A. S. (1988). Distributed programming with shared data. In *Proc. of the 1988 Int'l Conf. on Computer Languages*, pages 82–91.

Bennett, J. K., Carter, J. B., and Zwaenepoel, W. (1989). Munin: Shared memory for distributed memory multiprocessors. Technical Report COMP TR89-91, Dept. of Computer Science, Rice University.

Birman, K. P. (1993). The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53.

Birrell, A. and Nelson, B. (1984). Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59.

Birrell, A., Nelson, G., Owicki, S., and Wobber, E. (1993). Network objects. In *Proc. 14th ACM Symp. on Operating Systems Principles*.

Blau, B., Hughes, C. E., Moshell, M. J., and Lisle, C. (1992). Networked virtual environments. In *Proc. 1992 ACM Symp. on Interactive 3D Graphics*, pages 157–164.

Bricken, W. and Coco, G. (1994). The VEOS project. *Presence: Teleoperators and Virtual Environments*, 3(2):111–129.

Butz, A. (1997). Anymation with CATHI. In *Proceedings of AAAI/IAAI '97*, pages 957–962. AAAI Press.

Butz, A., Beshers, C., and Feiner, S. (1998). Of vampire mirrors and privacy lamps: Privacy management in multi-user augmented environments. In *Proc. ACM UIST '98*, pages 171–172, San Francisco, CA.

Calvin, J., Dickens, A., Gaines, B., Metzger, P., Miller, D., and Owen, D. (1993). The SIMNET virtual world architecture. In *Proc. IEEE VRAIS '93*, pages 450–455.

Cardelli, L. (1995). A language with distributed scope. *Computing Systems*, 8(1):27–59.

Cardelli, L. and Gordon, A. D. (1998). Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, pages 140–155.

Carlsson, C. and Hagsand, O. (1993). DIVE—a multi-user virtual reality system. In *Proc. IEEE VRAIS '93*, pages 394–400.

Carriero, N. and Gelernter, D. (1992). Linda in context. *Communications of the ACM*, 32(4):444–458.

Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. (1989). The amber system: Parallel programming on a network of multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP-12)*, pages 147–158.

Codella, C. F., Jalili, R., Koved, L., and Lewis, J. B. (1993). A toolkit for developing multi-user, distributed virtual environments. In *Proc. IEEE VRAIS '93*, pages 401–407.

Coulouris, G., Dollimore, J., and Kindberg, T. (1994). *Distributed Systems: Concepts and Design*. Addison Wesley.

Decouchant, D. (1986). Design of a distributed object manager for the Smalltalk-80 system. *ACM SIGPLAN Notices*, 21(11):444–444.

Dourish, P. (1996). *Open Implementation and Flexibility in CSCW Toolkits*. PhD thesis, University College Londen.

Elliott, C., Schechter, G., Yeung, R., and Abi-Ezzi, S. (1994). TBAG: A high level framework for interactive, animated 3D graphics applications. In *Computer Graphics (Proc. ACM SIGGRAPH '94)*, Annual Conference Series, pages 421–434.

Fairen, M. and Vinacua, A. (1997). Atlas, a platform for distributed graphics applications. In Arbab, F. and Slusallek, P., editors, *Proc. VI Eurographics Workshop on Programming Paradigms in Graphics*, pages 91–102.

Feiner, S., MacIntyre, B., Haupt, M., and Solomon, E. (1993a). Windows on the world: 2D windows for 3D augmented reality. In *Proc. ACM UIST '93*, pages 145–155.

Feiner, S., MacIntyre, B., Höllerer, T., and Webster, A. (1997). A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies*, 1(4):208–217.

Feiner, S., MacIntyre, B., and Seligmann, D. (1993b). Knowledge-based augmented reality. *Communications of the ACM*, 36(7):52–63.

Feiner, S. and Shamash, A. (1991). Hybrid user interfaces: Breeding virtually bigger interfaces for physically smaller computers. In *Proc. ACM UIST '91*, pages 9–17, Hilton Head, SC.

Feo, J.T., editor. (1992). A Comparative Study of Parallel Programming Languages: The Salishan Problems, Special Topics in Supercomputing, Volume 6, Elsevier Science Publishers, North-Holland.

Feiner, S., Webster, A., Krueger, T., MacIntyre, B., and Keller, E. (1995). Architectural anatomy. *Presence: Teleoperators and Virtual Environments*, 4(3):318–325.

Funkhouser, T. A. (1995). RING: A client-server system for multi-user virtual environments. In *Proc. 1995 ACM Symp. on Interactive 3D Graphics*, pages 85–92.

Gray, R. S. (1996). Agent Tcl: A flexible and secure mobile-agent system. In *4th Annual Tcl/Tk Workshop '96*, pages 9–23, Monterey, CA.

Grimsdale, G. (1991). dVS—distributed virtual environment system. In *Proc. Computer Graphics '91 Conference*.

Harbison, S. P. (1992). *Modula-3*. Prentice-Hall.

Holbrook, H., Singhal, S., and Cheriton, D. (1995). Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of SIGCOMM'95*, pages 328–341.

Holloway, R. (1991). *Trackerlib User's Maunal*. UNC Chapel Hill Computer Science Department.

IBM Corporation (1993). *IBM visualization Data Explorer*. IBM Corporation, Yorktown Heights, NY, fourth edition.

Janssen, B., Spreitzer, M., Larner, D., Jacobi, C. (1998). ILU Reference Manual. Xerox Palo Alto Research Center, Palo Alto, CA.

Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, 6(1):109–133.

Kazman, R. (1993). Making WAVES: On the design of architectures for low-end distributed virtual environments. In *Proc. IEEE VRAIS '93*, pages 443–449.

Kristensen, A. and Low, C. (1995). Problem-oriented object memory: Customizing consistency. In *Proc. ACM OOPSLA '95*, pages 399–413.

Levelt, W., Kaashoek, M., Bal, H., and Tanenbaum, A. (1992). A comparison of two paradigms for distributed shared memory. *Software Practice and Experience*, 22(11):985–1010.

Li, K. (1986). *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University.

Li, K. and Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359.

Liang, J., Shaw, C., and Green, M. (1991). On temporal-spatial realism in the virtual reality environment. In *Proc. ACM UIST '91*, pages 19–25.

Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312.

Lucas, B., Abram, G. D., Collins, N. S., Epstein, D. A., Gresh, D. L., and McAuliffe, K. P. (1992). An architecture for a scientific visualization system. In *Proc. Visualization '92*, pages 107–114, Boston, MA.

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzman, D. P., and Barham, P. T. (1995). Exploiting reality with multicast groups. *IEEE Computer Graphics and Applications*, 15(5):38–45.

Machiraju, V. (1997). A framework for migrating objects in distributed graphics applications. Masters dissertation, University of Utah, Department of Computer Science, Salt Lake City, UT.

MacIntyre, B. (1995). A testbed for distributed augmented reality systems. In *OOPSLA '95 Workshop on Reliability and Scalability in Distributed Object Systems*, Austin, TX.

MacIntyre, B. (1997). COTERIE: Columbia object-oriented toolkit for exploratory research in interactive environments. In *IEEE WETICE '97 Workshop on Distributed Systems Aspects of Sharing a Virtual Reality*, Cambridge, MA.

MacIntyre, B. and Feiner, S. (1994). New multimedia user interfaces: Virtual environments and ubiquitous computing. Technical Report Proc. Schloss Dagstuhl Seminar on Fundamentals and Perspectives on Multimedia Systems, Seminar No. 9427, Report No. 92, Schloss Dagstuhl, Germany.

MacIntyre, B. and Feiner, S. (1996a). Future multimedia user interfaces. *Multimedia Systems*, 4(5):250–268.

MacIntyre, B. and Feiner, S. (1996b). Language-level support for exploratory programming of distributed virtual environments. In *Proc. UIST '96*, pages 83–94, Seattle, WA.

MacIntyre, B. and Feiner, S. (1998). A distributed 3D graphics library. In *Computer Graphics (Proc. ACM SIGGRAPH '98)*, Annual Conference Series, pages 361–370, Orlando, FL.

MacIntyre, B. and Mynatt, E. (1998). Augmenting intelligent environments: Augmented reality as an interface to intelligent environments. In *Intelligent Environments Symposium, AAAI Spring Symposium Series*, Stanford University.

Manasse, M. S. (1993). The Trestle Toolkit. *The X Resource*, 5(1):107–112.

Manasse, M. S. (1995). The millicent protocols for electronic commerce. In *Proceedings of the First USENIX Workshop of Electronic Commerce*.

Mills, D. L. (1996). RFC 2030: Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI.

Mills, D. L. (1992). RFC 1305: Network time protocol (version 3) specification, implementation.

Najork, M. A. and Brown, M. H. (1995). Obliq-3D: A high-level, fast-turnaround 3D animation system. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):175–145.

Nog, S., Chawla, S., and Kotz, D. (1996). An RPC Mechanism for Transportable Agents. Technical Report PCS-TR96-280, Dartmouth College, Computer Science, Hanover, NH.

OMG (1992). *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Inc., Framingham, MA, 1.1 edition.

Open Communities (1997). The OpenCommunities Initiative. Information available at http://www.meitca.com/opencom.

Ousterhout, J. K. (1990). Tcl: An embeddable command language. In *USENIX Conference Proceedings*, pages 133–146.

Pausch, R., Burnette, T., Capehart, A., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., and White, J. (1995). Alice: A rapid prototyping system for 3D graphics. *IEEE Computer Graphics and Applications*, 15(3):8–11.

Perham, M., Smith, B. C., Janosi, T., and Lam, I. K. (1997). Redesigning Tcl-DP. In *5th Annual Tcl/Tk Workshop '97*, pages 49–53, Boston, MA.

Phillips, D., Pique, M., Moler, C., Torborg, J., and Greenberg, D. (1989). Distributed graphics: Where to draw the lines? SIGGRAPH 89 Panels, Boston, MA. Available at http://www.siggraph.org/publications/panels/siggraph89/.

Prakash, A. and Shim, H. S. (1994). DistView: Support for building efficient collaborative applications using replicated objects. In *Proc. ACM CSCW '94*, pages 153–162.

Rohlf, J. and Helman, J. (1994). IRIS performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Computer Graphics (Proc. ACM SIGGRAPH '94)*, Annual Conference Series, pages 381–394.

Roseman, M. and Greenberg, S. (1996). Building real-time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106.

Seligmann, D. D. and Feiner, S. (1991). Automated generation of intent-based 3D illustrations. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, pages 123–132.

Shaw, C. and Green, M. (1993). The MR toolkit peers package and experiment. In *Proc. IEEE VRAIS '93*, pages 18–22.

Shivers, O. (1994). A scheme shell. Technical Report MIT-LCS//MIT/LCS/TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science.

Singh, G., Serra, L., Png, W., Wong, A., and Ng, H. (1995). BrickNet: Sharing object behaviors on the net. In *Proc. IEEE VRAIS '95*, pages 19–25.

Sowizral, H., Rushforth, K., and Deering, M. (1998). *The Java 3D API Specification*. Addison Wesley, Reading, MA.

Stefik, M., Foster, G., Bobrow, D. G., Kahn, K., Lanning, S., and Suchman, L. (1987). Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47.

Strauss, P. S. and Carey, R. (1992). An object-oriented 3D graphics toolkit. In *Computer Graphics (Proc. ACM SIGGRAPH '92)*, Annual Conference Series, pages 341–349.

Sun Microsystems, Inc. (1998). The Java Shared Data Toolkit.

Tou, I., Berson, S., Estrin, G., Eterovic, Y., and Wu, E. (1994). Prototyping synchronous group applications. *IEEE Computer*, 27(5):48–56.

van Rossum, G. (1995). Python library reference. Technical Report CS-R9524, CWI - Centrum voor Wiskunde en Informatica.

Waters R.C, Anderson D.B., Barrus J.W., Brogan D.C., Casey M.A., McKeown S.G., Nitta

T., Sterns I.B., Yerazunis, W.S. (1997). Diamond Park and Spline: Social Virtual Reality with 3D Animation, Spoken Interaction, and Runtime Extendability, *Presence: Teleoperators and Virtual Environments*, 6(4):461--480.

Webster, A., Feiner, S., MacIntyre, B., Massie, B., and Krueger, T. (1996a). Augmented reality in architectural construction, inspection and renovation. In *Proc. ASCE Third Congress on Computing in Civil Engineering*, pages 913–919, Anaheim, CA.

Webster, A., Feiner, S., MacIntyre, B., Massie, W., and Krueger, T. (1996b). Augmented reality applications in architectural construction. In Bertol, D., editor, *Designing Digital Space: An Architect's Guide to Virtual Reality*, pages 193–200. John Wiley & Sons, New York, NY.

White, J. E. (1994). Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040.

Wollrath, A., Riggs, R., and Waldo, J. (1996). A distributed object model for the Java system. *Computing Systems*, 9(4):265–290.

Zeleznik, R. C., Conner, D. B., Wloka, M. M., Aliaga, D. G., Huang, N. T., Hubbard, P. M., Knep, B., Kaufman, H., Hughes, J. F., and van Dam, A. (1991). An object-oriented framework for the integration of interactive animation techniques. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, pages 105–112.

Zeleznik, R. C., Herndon, K. P., and Hughes, J. F. (1996). SKETCH: An interface for sketching 3D scenes. *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 163–170.

Zyda, M. J., Pratt, D. R., Monahan, J. G., and Wilson, K. P. (1992). NPSNET: Constructing a 3D virtual world. In *Proc. 1992 ACM Symp. on Interactive 3D Graphics*, pages 147–156.

APPENDIX A Example Generated Code

In Section 3.4.1, a detailed example of a Shared Object is presented. In order to use this shared object, a code generator is run at compile time, and six source files are created, which are presented here for the interested reader.

Section A.1 contains the implementation of the Shared Object, including the update event dispatch function (`ApplyUpdate_T`), the dispatch stubs (`Stub_*`), and the method wrappers (`Shared_*`). The file also contains the "pickling" routines that are used to marshal the object between sites. These pickling routines support the ability for a programmer to define a set of routines to read and write the object data. A default set of routines that marshall all the internal data fields are supplied. The object that programmers use to define custom marshalling routines is defined in the interface in Section A.6.

Sections A.2 and A.3 contain the module defining the Callback objects, used by programmers to receive notification of changes to an object instance. Sections A.4 and A.5 contain the interfaces defining proxy objects that can be used to embed both the Shared Object and its associated Callback object in an interpreted language such as Repo.

A.1 TrackerPositionSO.m3

```
(*****************************************************************
 * TrackerPositionSO.m3
 * DO NOT EDIT --> generated by shobjcodegen
 *                 Fri Aug 28 22:02:36 EDT 1998
 *****************************************************************)

MODULE TrackerPositionSO EXPORTS TrackerPositionPickle,
                                 TrackerPosition, TrackerPositionProxy;
IMPORT ThreadF, Rd, Tracker, SharedObjError, SharedObjStubLib,
       EventStubLib, SharedObjRep, TrackerPositionF, Wr,
TrackerPositionCB,
       EventProtocol, Event, PickleStubs, WeakRef, SharedObj, AtomList,
       EmbProxiedObj, Thread, Pickle2 AS Pickle, TrackerPosition,
       ObjectSpace;
```

```
CONST SharedObj_Protocol: EventProtocol.StubProtocol = 1;
EXCEPTION DuplicateSpecial;
TYPE T_SOMethods = {init, set};

REVEAL
  T = S BRANDED “Shared TrackerPosition.T v1.0” OBJECT
  OVERRIDES
    makeProxy := MakeProxy_T;
    applyUpdate := ApplyUpdate_T;

    init := Shared_init_T;
    set := Shared_set_T;
    get := Shared_get_T;
  END;

PROCEDURE MakeProxy_T (self: T) =
  BEGIN
    IF MkProxyT # NIL THEN
      MkProxyT(self);
    END;
  END MakeProxy_T;

PROCEDURE ApplyUpdate_T (self: T; ev: Event.T; h: EventStubLib.Handle)
  RAISES {SharedObj.Error, Event.Error, Rd.Failure, Thread.Alerted} =
  BEGIN
    IF ev.prot # SharedObj_Protocol THEN
      EventStubLib.RaiseUnmarshalFailure();
    END;
    WITH meth = SharedObjStubLib.InInt32(h) DO
      TRY
        SharedObjStubLib.AcquireWriteLock(self);
        self.updating := ThreadF.MyId();
        CASE meth OF
        | ORD(T_SOMethods.init) => Stub_init_T(self, h);
        | ORD(T_SOMethods.set) => Stub_set_T(self, h);
        ELSE
          EventStubLib.RaiseUnmarshalFailure();
        END;
      FINALLY
        self.updating := -1;
        SharedObjStubLib.ReleaseWriteLock(self);
      END;
    END;
  END ApplyUpdate_T;

PROCEDURE Shared_init_T(self: S): T RAISES {SharedObj.Error} =
  VAR out: SharedObjStubLib.Handle;
      id := ThreadF.MyId();
      dataPresent: BOOLEAN; <* NOWARN *>
  BEGIN
    (*************************************************)
    (* This get’s done once. After that, it’s a noop. *)
    (*************************************************)
    self := NARROW(SharedObj.Init(self), T);
    self.makeProxy();
    (*************************************************)
    IF NOT self.ok THEN SharedObjError.RaiseDeadObject() END;
    TRY
```

```
      SharedObjStubLib.AcquireReadLock(self);
      IF self.updating = id THEN
        (* do a simple, non-update call to the method *)
        RETURN S.init(self);
      END;
    FINALLY
      SharedObjStubLib.ReleaseReadLock(self);
    END;
    TRY
      out := SharedObjStubLib.StartCall(self);
      IF SharedObjStubLib.MarshalArgs(out) THEN
        SharedObjStubLib.OutInt32(out, ORD(T_SOMethods.init));
        END;
      SharedObjStubLib.SequenceCall(out, SharedObj_Protocol);
      TRY
        SharedObjStubLib.AcquireWriteLock(self);
        self.updating := id;
        Callback_pre_init_T(self);
        WITH res = S.init(self) DO
          Callback_post_init_T(self);
          RETURN res;
        END;
      FINALLY
        self.updating := -1;
        SharedObjStubLib.ReleaseWriteLock(self);
        SharedObjStubLib.EndCall(out);
      END;
    EXCEPT
    | Wr.Failure (ec) => SharedObjError.RaiseCommFailure(ec); <*ASSERT
FALSE*>
    | Thread.Alerted => SharedObjError.RaiseAlerted(); <*ASSERT FALSE*>
    END;
  END Shared_init_T;

PROCEDURE Shared_set_T(self: S; READONLY val_arg: Tracker.Report)
     RAISES {SharedObj.Error} =
  VAR out: SharedObjStubLib.Handle;
      id := ThreadF.MyId();
      dataPresent: BOOLEAN; <* NOWARN *>
  BEGIN
    IF NOT self.ok THEN SharedObjError.RaiseDeadObject() END;
    TRY
      SharedObjStubLib.AcquireReadLock(self);
      IF self.updating = id THEN
        (* do a simple, non-update call to the method *)
        S.set(self, val_arg);
        RETURN;
      END;
    FINALLY
      SharedObjStubLib.ReleaseReadLock(self);
    END;
    TRY
      out := SharedObjStubLib.StartCall(self);
      IF SharedObjStubLib.MarshalArgs(out) THEN
        SharedObjStubLib.OutInt32(out, ORD(T_SOMethods.set));
        SharedObjStubLib.OutRef(out, val_arg);
        END;
      SharedObjStubLib.SequenceCall(out, SharedObj_Protocol);
      TRY
```

```
        SharedObjStubLib.AcquireWriteLock(self);
        self.updating := id;
        Callback_pre_set_T(self, val_arg);
        S.set(self, val_arg);
        Callback_post_set_T(self, val_arg);
      FINALLY
        self.updating := -1;
        SharedObjStubLib.ReleaseWriteLock(self);
        SharedObjStubLib.EndCall(out);
      END;
    EXCEPT
    | Wr.Failure (ec) => SharedObjError.RaiseCommFailure(ec);
    | Thread.Alerted => SharedObjError.RaiseAlerted();
    END;
  END Shared_set_T;

PROCEDURE Shared_get_T(self: S): Tracker.Report RAISES {SharedObj.Error,
    Thread.Alerted} =
  BEGIN
    IF NOT self.ok THEN SharedObjError.RaiseDeadObject() END;
    TRY
      SharedObjStubLib.AcquireReadLock(self);
      RETURN S.get(self);
    FINALLY
      SharedObjStubLib.ReleaseReadLock(self);
    END;
  END Shared_get_T;

PROCEDURE Stub_init_T(self: S; <* NOWARN *> in: EventStubLib.Handle)
     RAISES {SharedObj.Error} =
  BEGIN
    Callback_pre_init_T(self);
    EVAL S.init(self);
    Callback_post_init_T(self);
  END Stub_init_T;

PROCEDURE Stub_set_T(self: S; <* NOWARN *> in: EventStubLib.Handle)
     RAISES {SharedObj.Error, Rd.Failure, Thread.Alerted} =
    VAR val_arg: Tracker.Report;
        dataPresent: BOOLEAN <* NOWARN *>;
  BEGIN
    val_arg := SharedObjStubLib.InRef(in, TYPECODE(Tracker.Report));
    Callback_pre_set_T(self, val_arg);
    S.set(self, val_arg);
    Callback_post_set_T(self, val_arg);
  END Stub_set_T;

PROCEDURE Callback_pre_init_T(self: T) =
  VAR cbs := self.callbacks;
  BEGIN
    WHILE cbs # NIL DO
      IF cbs.head.ready THEN
        WITH ref = WeakRef.ToRef(cbs.head.weakRef) DO
          IF ref # NIL THEN
            WITH cb = NARROW(ref, TrackerPositionCB.T) DO
              IF NOT cb.pre_init(self) THEN
                cb.pre_anyChange(self);
              END;
            END;
```

```
          END;
        END;
      END;
      cbs := cbs.tail;
    END;
  END Callback_pre_init_T;

PROCEDURE Callback_post_init_T(self: T) =
  VAR cbs := self.callbacks;
  BEGIN
    WHILE cbs # NIL DO
      IF cbs.head.ready THEN
        WITH ref = WeakRef.ToRef(cbs.head.weakRef) DO
          IF ref # NIL THEN
            WITH cb = NARROW(ref, TrackerPositionCB.T) DO
              IF NOT cb.post_init(self) THEN
                cb.post_anyChange(self);
              END;
            END;
          END;
        END;
      END;
      cbs := cbs.tail;
    END;
  END Callback_post_init_T;

PROCEDURE Callback_pre_set_T(self: T; READONLY val_arg: Tracker.Report)
=
  VAR cbs := self.callbacks;
  BEGIN
    WHILE cbs # NIL DO
      IF cbs.head.ready THEN
        WITH ref = WeakRef.ToRef(cbs.head.weakRef) DO
          IF ref # NIL THEN
            WITH cb = NARROW(ref, TrackerPositionCB.T) DO
              IF NOT cb.pre_set(self, val_arg) THEN
                cb.pre_anyChange(self);
              END;
            END;
          END;
        END;
      END;
      cbs := cbs.tail;
    END;
  END Callback_pre_set_T;

PROCEDURE Callback_post_set_T(self: T; READONLY val_arg: Tracker.Report)
=
  VAR cbs := self.callbacks;
  BEGIN
    WHILE cbs # NIL DO
      IF cbs.head.ready THEN
        WITH ref = WeakRef.ToRef(cbs.head.weakRef) DO
          IF ref # NIL THEN
            WITH cb = NARROW(ref, TrackerPositionCB.T) DO
              IF NOT cb.post_set(self, val_arg) THEN
                cb.post_anyChange(self);
              END;
            END;
```

```
          END;
        END;
      END;
      cbs := cbs.tail;
    END;
  END Callback_post_set_T;

(* The pickling routine for this shared object. We will register a
   pickler for TrackerPosition.S, and then handle both S and T.
   Pickling subtypes of T is illegal. *)
REVEAL
  TSpecial = SharedObj.Special BRANDED "TrackerPosition.TSpecial" OBJECT
             OVERRIDES
               write := DefaultSpWrite_T;
               read := DefaultSpRead_T;
             END;

TYPE
  T_Special = Pickle.Special OBJECT
                mu: MUTEX;
                sp: TSpecial;
                registered: BOOLEAN := FALSE;
              OVERRIDES
                write := Write_T;
                read := Read_T;
              END;

PROCEDURE DefaultSpWrite_T (<*UNUSED*>self: TSpecial; shobj:
SharedObj.T;
                            out: Pickle.Writer)
                            RAISES {Pickle.Error, Wr.Failure,
Thread.Alerted} =
  VAR
    obj := NARROW(shobj, S);
  BEGIN
    PickleStubs.OutRef(out, obj.data);

  END DefaultSpWrite_T;

PROCEDURE Write_T (<*UNUSED*>ts: T_Special; ref: REFANY;
                   out: Pickle.Writer)
                   RAISES {Pickle.Error, Wr.Failure, Thread.Alerted} =
  VAR
    obj: S;
    sp: TSpecial;
    tc := TYPECODE(ref);
  BEGIN
    IF tc # TYPECODE(S) AND tc # TYPECODE(T) THEN
      RAISE Pickle.Error("Can't pickle subtypes of TrackerPosition.T");
    END;
    obj := NARROW(ref, S);
    out.writeType(tc);
    SharedObjStubLib.StartWritePickle(obj, out);
    LOCK spT.mu DO
      sp := spT.sp;
    END;
    sp.write(obj, out);
    SharedObjStubLib.EndWritePickle(obj, out);
  END Write_T;
```

```
PROCEDURE DefaultSpRead_T (<*UNUSED*>self: TSpecial; shobj: SharedObj.T;
                           in: Pickle.Reader) RAISES {
  Pickle.Error, Rd.EndOfFile, Rd.Failure, Thread.Alerted} =
  VAR
    obj := NARROW(shobj, S);
  BEGIN
    obj.data := PickleStubs.InRef(in, TYPECODE(Tracker.Report));

  END DefaultSpRead_T;

PROCEDURE Read_T (<*UNUSED*>ts: T_Special; in: Pickle.Reader;
                  id: Pickle.RefID):REFANY RAISES {
  Pickle.Error, Rd.EndOfFile, Rd.Failure, Thread.Alerted} =
  VAR
    space: ObjectSpace.T;
    obj: S;
    sp: TSpecial;
    proxy: EmbProxiedObj.Proxy;
    tc := in.readType();
  BEGIN
    IF tc = TYPECODE(T) THEN
      obj := NEW(T);
    ELSIF tc = TYPECODE(S) THEN
      obj := NEW(S);
    ELSE
     RAISE Pickle.Error("Can't unpickle subtypes of TrackerPosition.T");
    END;
    space := in.read();
    SharedObjStubLib.StartReadPickle(obj, in, space);
    LOCK spT.mu DO
      sp := spT.sp;
    END;
    sp.read(obj, in);
    IF tc = TYPECODE(T) THEN
      obj := SharedObjStubLib.SetupNewCopy(obj, in, id, space);
      proxy := PickleStubs.InRef(in);
      IF obj.proxy = NIL THEN
        obj.proxy := proxy;
      END;
      obj.makeProxy();
    ELSE
      obj.proxy := NIL;
      obj.proxy := PickleStubs.InRef(in);
    END;
    RETURN obj;
  END Read_T;

PROCEDURE RegisterSpecial_T (sp: TSpecial) =
  <* FATAL DuplicateSpecial *>
  BEGIN
    (* we will need to NEW it here if RegisterSpecial_T
       is called from TrackerPosition *)
    IF spT = NIL THEN
      spT := NEW(T_Special, sc := TYPECODE(S), mu := NEW(MUTEX));
    END;
    LOCK spT.mu DO
      IF spT.registered THEN
        RAISE DuplicateSpecial;
```

```
      END;
      spT.registered := TRUE;
      spT.sp := sp;
    END;
  END RegisterSpecial_T;

VAR
  spT: T_Special := NIL;

BEGIN
  IF spT = NIL THEN
    spT := NEW(T_Special,
       sc := TYPECODE(S),
      mu := NEW(MUTEX),
      sp := NEW(TSpecial));
  END;
  Pickle.RegisterSpecial(spT);
END TrackerPositionSO.
```

A.2 TrackerPositionCB.i3

```
(*****************************************************************
 * TrackerPositionCB.i3
 * DO NOT EDIT --> generated by shobjcodegen
 *                 Fri Aug 28 22:02:36 EDT 1998
 *****************************************************************)

INTERFACE TrackerPositionCB;

IMPORT Tracker, SharedObj, TrackerPosition;

TYPE
  T <: PublicT;
  PublicT = SharedObj.Callback OBJECT
    METHODS
      init (obj: TrackerPosition.T): T;
      cancel ();
      pre_anyChange (READONLY obj: TrackerPosition.T);
      post_anyChange (READONLY obj: TrackerPosition.T);
      pre_init (READONLY obj: TrackerPosition.T): BOOLEAN;
      post_init (READONLY obj: TrackerPosition.T): BOOLEAN;
      pre_set (READONLY obj: TrackerPosition.T;
               READONLY val: Tracker.Report): BOOLEAN;
      post_set (READONLY obj: TrackerPosition.T;
                READONLY val: Tracker.Report): BOOLEAN;
    END;

END TrackerPositionCB.
```

A.3 TrackerPositionCB.m3

```
(*****************************************************************
 * TrackerPositionCB.m3
 * DO NOT EDIT --> generated by shobjcodegen
 *                 Fri Aug 28 22:02:36 EDT 1998
```

```
 ***************************************************************)

MODULE TrackerPositionCB EXPORTS TrackerPositionCB,
TrackerPositionCBProxy;
IMPORT Tracker, SharedObjStubLib, SharedObjRep, WeakRefListFuncs,
       WeakRefList, WeakRef, TrackerPosition, WeakerRef;

REVEAL
  T = PublicT BRANDED OBJECT
    obj: TrackerPosition.T;
    wref: WeakerRef.T;
  OVERRIDES
    init := Init_T;
    cancel := Cancel_T;
    pre_anyChange := Pre_anyChange_T;
    post_anyChange := Post_anyChange_T;
    pre_init := Pre_init_T;
    post_init := Post_init_T;
    pre_set := Pre_set_T;
    post_set := Post_set_T;
  END;

PROCEDURE Init_T (self: T; obj: TrackerPosition.T): T =
  VAR
    wref := NEW(WeakerRef.T,
                weakRef := WeakRef.FromRef(self, Cleanup_T_CB),
                ready := TRUE);
  BEGIN
    self.obj := obj;
    self.wref := wref;
    IF MkProxyTCB # NIL AND self.proxy = NIL THEN
      MkProxyTCB (self);
    END;
    SharedObjStubLib.AcquireWriteLock(obj);
    TRY
      obj.callbacks := WeakRefList.Cons(wref, obj.callbacks);
    FINALLY
      SharedObjStubLib.ReleaseWriteLock(obj);
    END;
    RETURN self;
  END Init_T;

PROCEDURE Cancel_T (self: T) =
  BEGIN
    SharedObjStubLib.AcquireWriteLock(self.obj);
    TRY
      EVAL WeakRefListFuncs.DeleteD(self.obj.callbacks, self.wref);
    FINALLY
      SharedObjStubLib.ReleaseWriteLock(self.obj);
    END;
  END Cancel_T;

PROCEDURE Cleanup_T_CB (READONLY wref: WeakRef.T; ref: REFANY) =
  VAR
    cb := NARROW(ref, T);
    weakerRef := NEW(WeakerRef.T, weakRef := wref);
  BEGIN
    SharedObjStubLib.AcquireWriteLock(cb.obj);
    TRY
```

```
      (* Callback is gone, so delete it *)
      EVAL WeakRefListFuncs.DeleteD(cb.obj.callbacks, weakerRef);
    FINALLY
      SharedObjStubLib.ReleaseWriteLock(cb.obj);
    END;
  END Cleanup_T_CB;

PROCEDURE Pre_anyChange_T (self: T; READONLY obj: TrackerPosition.T) =
  BEGIN
    (* Default calls proxy or does nothing. *)
    IF self.proxy # NIL THEN
      NARROW (self.proxy, CBProxyT).pre_anyChange (obj);
    END;
  END Pre_anyChange_T;

PROCEDURE Post_anyChange_T (self: T; READONLY obj: TrackerPosition.T) =
  BEGIN
    (* Default calls proxy or does nothing. *)
    IF self.proxy # NIL THEN
      NARROW (self.proxy, CBProxyT).post_anyChange (obj);
    END;
  END Post_anyChange_T;

PROCEDURE Pre_init_T (self: T; READONLY obj: TrackerPosition.T): BOOLEAN
=
  BEGIN
    (* Default calls proxy or does nothing. *)
    IF self.proxy # NIL THEN
      RETURN NARROW (self.proxy, CBProxyT).pre_init (obj);
      END;
    RETURN FALSE;
  END Pre_init_T;

PROCEDURE Post_init_T (self: T; READONLY obj: TrackerPosition.T):
BOOLEAN =
  BEGIN
    (* Default calls proxy or does nothing. *)
    IF self.proxy # NIL THEN
      RETURN NARROW (self.proxy, CBProxyT).post_init (obj);
      END;
    RETURN FALSE;
  END Post_init_T;

PROCEDURE Pre_set_T (self: T; READONLY obj: TrackerPosition.T;
                     READONLY val: Tracker.Report): BOOLEAN =
  BEGIN
    (* Default calls proxy or does nothing. *)
    IF self.proxy # NIL THEN
      RETURN NARROW (self.proxy, CBProxyT).pre_set (obj, val);
      END;
    RETURN FALSE;
  END Pre_set_T;

PROCEDURE Post_set_T (self: T; READONLY obj: TrackerPosition.T;
                      READONLY val: Tracker.Report): BOOLEAN =
  BEGIN
    (* Default calls proxy or does nothing. *)
    IF self.proxy # NIL THEN
      RETURN NARROW (self.proxy, CBProxyT).post_set (obj, val);
```

```
      END;
    RETURN FALSE;
  END Post_set_T;

BEGIN
SharedObjStubLib.InhibitTransmission(TYPECODE(T), "default T callback
cannot be transmitted/duplicated");
END TrackerPositionCB.
```

A.4 TrackerPositionProxy.i3

```
(*****************************************************************
 * TrackerPositionProxy.i3
 * DO NOT EDIT --> generated by shobjcodegen
 *                 Fri Aug 28 22:02:36 EDT 1998
 *****************************************************************)

INTERFACE TrackerPositionProxy;

IMPORT TrackerPosition;

VAR
  MkProxyT : PROCEDURE(x: TrackerPosition.T) := NIL;

END TrackerPositionProxy.
```

A.5 TrackerPositionCBProxy.i3

```
(*****************************************************************
 * TrackerPositionCBProxy.i3
 * DO NOT EDIT --> generated by shobjcodegen
 *                 Fri Aug 28 22:02:36 EDT 1998
 *****************************************************************)

INTERFACE TrackerPositionCBProxy;

IMPORT Tracker, TrackerPositionCB, EmbProxiedObj, TrackerPosition;

VAR
  MkProxyTCB : PROCEDURE(x: TrackerPositionCB.T) := NIL;

TYPE
  CBProxyT = EmbProxiedObj.Proxy OBJECT METHODS
    pre_anyChange (READONLY obj: TrackerPosition.T);
    post_anyChange (READONLY obj: TrackerPosition.T);
    pre_init (READONLY obj: TrackerPosition.T): BOOLEAN;
    post_init (READONLY obj: TrackerPosition.T): BOOLEAN;
    pre_set (READONLY obj: TrackerPosition.T;
             READONLY val: Tracker.Report): BOOLEAN;
    post_set (READONLY obj: TrackerPosition.T;
              READONLY val: Tracker.Report): BOOLEAN;
  END;

END TrackerPositionCBProxy.
```

A.6 TrackerPositionPickle.i3

```
(*****************************************************************
 * TrackerPositionPickle.i3
 * DO NOT EDIT --> generated by shobjcodegen
 *                 Fri Aug 28 22:02:36 EDT 1998
 *****************************************************************)

INTERFACE TrackerPositionPickle;

IMPORT SharedObj;

TYPE
TSpecial <: SharedObj.Special;
PROCEDURE RegisterSpecial_T(sp: TSpecial);
END TrackerPositionPickle.
```

APPENDIX B Tracker Modules

In this appendix, we present the Repo help files for the various modules related to the tracker system, first discussed in Section 2.5.1 and used as the basis of the examples in Section 3.4.1, Section 4.6.1 and Section 5.4.2.

B.1 The Basic Modules

B.1.1 Kalman

```
   Kalman_New(): T
   Kalman_Filter(T, Quaternion): Quaternion
WHERE
   Kalman <: EmbProxiedObj &
      { filter: (Quaternion) => Quaternion }
```

This module implements a simple Kalman predictive filter, based on the approach used in [Liang et al., 1991].

B.1.2 Tracker

```
   Tracker_EndOfFile: Exception
   Tracker_Error: Exception
   Tracker_NewReport(ReportProto): Tracker_Report
   Tracker_NewReport2D(Report2DProto): Tracker_Report2D
   Tracker_NewReport3D(Report3DProto): Tracker_Report3D
   Tracker_GetTimestamp(Tracker_Report): Real
   Tracker_GetButtons(Tracker_Report, Buttons): Ok
   Tracker_GetPosition2D(Tracker_Report2D): Point2
   Tracker_GetPosition3D(Tracker_Report3D): Point3
   Tracker_GetPosition3DError(Tracker_Report3D): Real
   Tracker_GetOrientationM(Tracker_Report3D): Matrix4
   Tracker_GetOrientationQ(Tracker_Report3D): Quaternion
   Tracker_GetOrientationError(Tracker_Report3D): Real
WHERE
```

```
  Tracker <: EmbProxiedObj &
    { calculateStatistics: () => Text,
      reset: () => Ok ! Tracker_Error thread_alerted,
      read: () => TrackerReport ! Tracker_EndOfFile Tracker_Error
                    thread_alerted,
      close: () => Ok ! Tracker_Error }

  Tracker_Report <: EmbProxiedObj
  Tracker_Report2D <: Tracker_Report
  Tracker_Report3D <: Tracker_Report
  ReportProto = { ts: Real, buttons: Buttons }
  Report2DProto = ReportObj & { pos: Point2 }
  Report3DProto = ReportObj & { pos: Point3, pos_error: Real,
                                orientation: Orientation,
                                orientation_error: Real }
  Buttons = [n*Bool]
  Point2 = [2*Int]
  Orientation: Quaternion & Matrix4
```

This is the abstract Tracker module. It defines the basic methods that must be supported by the concrete trackers in Section B.2, and defines the basic Tracker Report, as well as the prototype 2D and 3D Reports used by the concrete trackers.

B.1.3 TrackerPosition

```
  TrackerPosition_New(): T;
WHERE
  T <: SharedObj_T & { init: () => T ! SharedObj_Error Thread_alerted,
                       get: () => Tracker_Report ! SharedObj_Error
                                  Thread_alerted,
                       set: (Tracker_Report) => Ok ! SharedObj_Error
                                  Thread_alerted }
```

This module exposes the replicated Tracker Position object, discussed in Section 3.4.1, into Repo.

B.1.4 TrackerPositionCB

```
  TrackerPositionCB_New(obj: TrackerPosition_T, overrides: Obj): T;
  TrackerPositionCB_Cancel(cbobj: T): T;
WHERE
  T <: SharedObj_T & overrides;
  overrides contains one or more of these callback methods:
    pre`init(obj: TrackerPosition_T): bool;
    post`init(obj: TrackerPosition_T): bool;
    pre`set(obj: TrackerPosition_T, val: Tracker_Report): bool;
    post`set(obj: TrackerPosition_T, val: Tracker_Report): bool;
```

```
    pre`anyChange(obj: TrackerPosition_T);
    post`anyChange(obj: TrackerPosition_T);
```

This module exposes the replicated Tracker Position Callback object, discussed in Section 3.4.1, into Repo.

B.1.5 TrackerServer

```
  TrackerServer_New(Tracker): T;
  TrackerServer_NewLatest(Tracker): T;
  TrackerServer_NewLatestMulti(Tracker): Multi;
WHERE
  T <: SharedObj_T &
       {  add: (TrackerPosition) => Ok,
          remove: (TrackerPosition) => Ok,
          start: () => Ok,
          stop: () => Ok }
  Multi <: T &
       { addSingle: (TrackerPosition, index) => Ok }
```

This module defines a simple “tracker server,” which is a thread that continuously reads from a tracker and stores the Reports read in one or more Tracker Position objects. There are three forms of the server: the basic one (created with `TrackerServer_New`), one that only reads the latest value if it cannot read the reports as fast as the tracker hardware produces them (created with `TrackerServer_NewLatest`), and one that reads from a multi-device tracker (such as the Flock of Birds in Section B.2.2 or the DynaSight in Section B.2.1).

B.2 The Tracking Device Modules

This section contains the help files for each of the concrete tracking devices we currently support.

B.2.1 Dynasight

```
  Dynasight_Open(path: text; mode: Mode): T ! Tracker_Error
  Dynasight_GetStatus(Dynasight_Report): SensorStatus
  Dynasight_GetSync(Dynasight_Report): Bool
  Dynasight_GetTargetNumber(Dynasight_Report): Int
```

```
WHERE
   Dynasight <: Tracker
   Dynasight_Report <: Tracker_Report3D
   SensorStatus = Text   (one of "Search", "Coast", "Caution", "Track")
   Mode = Text ("Passive", "ATA1", "ATA2", "ATA3", "ATA4",
               "ATA3T", "ATA4T", "ATA4Y2", or "ATA4Y3")
   There are no buttons
```

The Origin Instruments DynaSight optical radar system.

B.2.2 FOB

```
   FOB_Open(path: Text, num: Int, fast: Bool): T ! thread_alerted
        Tracker_Error
   FOB_GetTargetNumber(FOB_Report): Int
WHERE
   FOB <: Tracker &
     { demandReporting: () => Ok ! thread_alerted Tracker_Error,
       streamReporting: () => Ok ! thread_alerted Tracker_Error,
       enableTransmitter: () => Ok ! thread_alerted Tracker_Error,
       disableTransmitter: () => Ok ! thread_alerted Tracker_Error,
       flipHemisphere: (unit: Int) => Ok ! thread_alerted Tracker_Error,
       setHemisphere: (hemi: Hemi) => Ok ! thread_alerted Tracker_Error
     }
   FOB_Report <: Tracker_Report3D
   Hemi = Text ("Foward", "Aft", "Left", "Right", "Upper", "Lower")
```

The Ascension Technologies Flock of Birds magnetic tracking system.

B.2.3 Logitech

```
   Logitech_Open(path: text): T ! thread_alerted Tracker_Error
   Logitech_GetStatus(Logitech_Report): SensorStatus
   Logitech_GetButtons(Logitech_Report, Buttons): Ok
WHERE
   Logitech <: Tracker &
     { demandReporting: () => Ok ! thread_alerted Tracker_Error,
       streamReporting: () => Ok ! thread_alerted Tracker_Error,
       enableTransmitter: () => Ok ! thread_alerted Tracker_Error,
       disableTransmitter: () => Ok ! thread_alerted Tracker_Error,
       incrementalReporting: () => Ok ! thread_alerted Tracker_Error,
       setFilterCount: (count: FilterCount) => Ok ! thread_alerted
                                   Tracker_Error }
   FilterCount = 0...10
   Logitech_Report <: Tracker_Report3D
   SensorStatus = Text   (one of "Fringe", "Out", "Track")
   Buttons = { left: Bool, right: Bool, middle: Bool, suspend: Bool }
```

The Logitech 6DOF ultrasonic tracking system.

B.2.4 MSMouse

```
   MSMouse_Open(path: text): T ! Tracker_Error
   MSMouse_GetButtons(MSMouse_Report, Buttons): Ok
WHERE
   MSMouse <: Tracker
   MSMouse_Report <: Tracker_Report2D
   Buttons = { left: Bool, right: Bool, middle: Bool}
```

The Microsoft 2D 3-button mouse.

B.2.5 PTU

```
   PTU_Open(path: text): T ! thread_alerted Tracker_Error
WHERE
   PTU <: Tracker &
     { hardReset: () => Ok ! Tracker_Error thread_alerted,
       awaitExecution: () => Ok ! thread_alerted Tracker_Error,
       moveAbsPanAngle: (pan: Real) => Bool ! thread_alerted
                                Tracker_Error,
       moveAbsTiltAngle: (tilt: Real): => Bool ! thread_alerted
                                Tracker_Error,
       moveAbsAngle: (pan, tilt: Real) => Bool ! thread_alerted
                                Tracker_Error,
       moveOffsetAngle: (pan, tilt: Real)=> Bool ! thread_alerted
                                Tracker_Error,
       haltPan: () => Ok ! thread_alerted Tracker_Error,
       haltTilt: () => Ok ! thread_alerted Tracker_Error,
       haltAll: () => Ok ! thread_alerted Tracker_Error,
       getPanRange: () => [Real,Real],
       getTiltRange: () => [Real,Real]
     }

   PTU_Report <: Tracker_Report3D

   There are no buttons
```

The Directed Perception 3DOF Pan/Tilt unit.

B.2.6 RingMouse

```
   RingMouse_Open(path: text): T ! Tracker_Error
   RingMouse_GetStatus(RingMouse_Report): SensorStatus
```

```
  RingMouse_GetButtons(RingMouse_Report, Buttons): Ok
WHERE
  RingMouse <: Tracker
  RingMouse_Report <: Tracker_Report3D
  Buttons = Tracker_Buttons & { left: Bool, right: Bool}
  SensorStatus = Text   (one of "Sleep", "Track")
```

The Kantek Spectrum RingMouse ultrasonic 3DOF position tracker.

B.2.7 Scanner

```
  Scanner_Open(path: text): T ! Tracker_Error
  Scanner_GetBarcode(Scanner_Report): Text
WHERE
  Scanner <: Tracker
  Scanner_Report <: Tracker_Report
  There are no buttons
```

The PSC Inc. QuickScan barcode scanner.

B.2.8 Trimble

```
  Trimble_Open(path: text, out: Wr|Ok): T ! thread_alerted Tracker_Error
  Trimble_GetHealth(Trimble_Report): Health
  Trimble_GetVelocity(Trimble_Report): Velocity
  Trimble_GetVersion(Trimble_Report): Real
  Trimble_GetSatellites(Trimble_Report): Satellites
  Trimble_GetMessage(Trimble_Report): SystemMessage
  Trimble_GetLLA(Trimble_Report): LLA
WHERE
  Trimble <: Tracker;
  Trimble_Report <: Tracker_Report3D;

  Satellites = {number: Int, used: [Int,Int,Int,Int], ts: Real};
  Velocity = {x y z: Real, xyzts: Real,
              east north up: Real, enuts: Real};
  SystemMessage = {severeFailureReport message: Text, ts: Real};
  Health = {status: Text, statusCode: Int,
         batteryBackup antennaStatus timeClockStatus
         atoDConverterStatus almanacStatus: BOOLEAN, ts: Real};
  LLA = {longitude latitude altitude relativeLong relativeLat: Real};
```

The Trimble GPS 3DOF position tracker.

B.2.9 vIO

```
   vIO_Open(path: text): T ! thread_alerted Tracker_Error
WHERE
   vIO <: Tracker &
     { demandReporting: () => Ok ! thread_alerted Tracker_Error,
       streamReporting: () => Ok ! thread_alerted Tracker_Error,
       setAngleMode: (mode: Mode) => Ok ! thread_alerted Tracker_Error }
   vIO_Report <: Tracker_Report3D

   Mode = Text   (one of "Tilt", "Yaw", "All")
   There are no buttons
```

The Virtual I/O 3DOF orientation tracker.

APPENDIX C Repo Syntax

This Appendix contains a summary of Repo's syntax. This is based on (and is very similar to) the Obliq syntax summary in [Cardelli, 1995].

```
TOP-LEVEL PHRASES                 any term or definition ended by ";"
  a;

DEFINITIONS (identifiers are denoted by "x", terms are denoted by "a")
  let x1=a1,...,xn=an             definition of constant identifiers
  let rec x1=a1,...,xn=an         definition of recursive procedures
  var x1=a1,...,xn=an             definition of updatable identifiers
  var replicated x1=a1,...,xn=an  definition of replicated updatable ids
  var simple x1=a1,...,xn=an      definition of simple updatable ids

SEQUENCES (denoted by "s")        each "ai" (a term or a definition) is
  a1;...;an                       executed; yields "an" (or "ok" if n=0)

TERMS (denoted by "a","b","c";    identifiers are denoted by "x","l";
                                  modules are denoted by "m")
  xm_x                            identifiers
  x:=a                            assignment

  ok true false 'a' "abc" 3 1.5   constants

  [a1,...,an]                     arrays
  replicated [a1,...,an]          replicated arrays
  simple [a1,...,an]              simple arrays
  a[b]a[b]:=c                     array selection, array update
  a[b for b']a[b for b']:=c        subarray selection, subarray update

  option "l" => s end             term "s" tagged by "l"

  proc(x1,...,xn) s end           procedures
  a(b1,...,bn)                    procedure invocation
  m_x(a1,...,an)                  invocation of "x" from module "m"
  a b c                           infix (right-ass.) version of "b(a,c)"

  meth(x,x1,...,xn) s end         method with self "x"
  umeth(x,x1,...,xn) s end        replicated object update method
  {l1=>a1,...,ln=>an}             object with fields named "l1"..."ln"
  {protected, serialized, ...}    protected and serialized object
  {simple, ...}                   simple object
  {replicated, ...}               replicated object
  {l1=>alias l2 of a2 end,...}    object with delegated fields
```

```
a.l    a.l(a1, ..., an)        field selection / method invocation
a.l:=b                         field update / method override
clone(a1,...,an)               object cloning
replicated(a,umethlist)        replicated clone of object "a"
replicated(a)                  replicated copy of array "a"
simple(a)                      simple clone/copy of object/array "a"
remote(a)                      remote clone/copy of object/array "a"
a1.l1:=alias l2 of a2 end      field delegation
delegate a1 to a2 end          object delegation
unreachable a1 do a2           unreachable data value notification
objectpickler a1 reader a2 writer a3
                               pickle a1 using a2 for reading and
                               a3 for writing
addhelp m sort "s1" short "s2" full "s3"
                               setup help entry for m

d                              definition
if s1 then s2                  conditional
  elsif s3 then s4... else sn end  ("elsif", "else" optional)
a andif ba orif b              conditional conjunction/disjunction
case s of "l1"(x1,m1)=>s1,...,  case over the tag "li" of an option
  "ln"(xn,m1)=>snbinding "xi" in "si" ("mi" optional)
  else s0 end                  ("else" optional)
                               of match subexpressions of "li"
loop s end                     loop
for i=a to b do s end          iteration through successive integers
foreach i in a do s end        iteration through an array
foreach i in a map s end       yielding an array of the results
exit                           exit the innermost loop, for, foreach

exception("exc")               new exception value named "exc"
raise(a)                       raise an exception
try s except                   exception capture
  a1=>s1,...,an=>sn else s0 end ("else" optional)
try s1 finally s2 end          finalization

condition() signal(a) broadcast(a)
                               creating and signaling a condition
watch s1 until s2 end          waiting for a signal and a boolean
                               guard
fork(a1,a2)   join(a)          forking and joining a thread
pause(a)                       pausing the current thread

mutex()                        creating a mutex
lock s1 do s2 end              locking a mutex in a scope
wait(a1,a2)                    waiting on a mutex for a condition

(s)                            block structure / precedence group
```

APPENDIX D

Additional Enhancements to Repo

In addition to the changes to the language syntax and semantic required by the addition of replicated data, there are a number of other enhancements in Repo. These changes were made to support exploratory programming of distributed interactive applications, usually in response to a specific need or problem we encountered while developing our prototypes.

D.1 Additional Syntax Changes

At the beginning of Section 4.4, we mentioned that there was one change we made to the Repo syntax that is not compatible with Obliq. That change is the addition of regular expression support to Obliq's `case` statement, which is used in conjunction with Obliq's *option* values. Options are created by associating an arbitrary value with a textual tag, using the following statement:

```
option tag => value end
```

This statement returns an option value that can be used in a case statement, the syntax of which is:

```
case o of
   l1(x1)=> s1,..., ln(xn)=> sn else s0 end
```

Given an option `o`, if one of the labels l_i exactly matches `o`'s tag, the corresponding statement s_i is executed. If the optional variable name x_i is supplied, `o`'s value is assigned to it in the context of s_i. If no labels match, the else statement is executed.

However, the tags are not arbitrary text strings, but follow the same guidelines as identifiers (i.e., variable names). We found that the options and case statement were not very useful when defined this way. In particular, most of the time we found ourselves wanting a case statement, we wanted to be able to select between arbitrary text strings, and we wanted to be able to partially match these strings using regular expressions.

Therefore, we changed the syntax of these two statements to use text strings for the tags and case labels, and added support for regular expressions to the case labels so that the labels do not have to match the option tag exactly. The regular expressions follow the Unix *regex* syntax, including supporting substring matching using the "( )" syntax. We added a second optional variable name y_i that, if present, will be the name of a variable to contain an array describing the matched substrings in the text string (the full match followed by the substring matches, listed as integer [start,end] pairs):

```
option "tag string" => value end

case o of
   l1(x1,y1)=> s1,..., ln(xn,yn)=> sn else s0 end
```

We make extensive use of the new case statement in our code, such as in the object directories of Section 4.6.5 or the Sketch example of Section 6.2.

Another change we made to the Repo syntax was to add the `unreachable` statement. This statement provides notification when a local reference to a client-server object becomes invalid because the network address of the object can no longer be contacted, either because the process has terminated, or there is a problem with the network. The

`unreachable` statement takes a procedure argument that will be executed when the system determines that the object is unreachable.

```
unreachable object do notification-proc
```

Here is an example of this statement in use, taken from the enhanced version of the replicated mutex example in Appendix E (the simple version of the distributed mutex is presented in Section 4.6.4):

```
unreachable id do
     proc (o,st)
        try
           s.dequeueId(localId, localId.txt);
        except unheld => end;
     end;
```

In this example, when the object `id` becomes unreachable, the method `s.dequeueId` is called. See the appendix for a more in depth discussion.

The other changes to Repo's syntax are minor enhancements to the way modules are defined, aimed at supporting the creation of more complex programs. This include the ability to define on-line help files for Repo modules (the syntax can be seen in Figure 4-8) and to hide information inside modules. Previously, only built-in modules could have on-line help, and any values defined inside a module could be accessed from outside of it. We will not detail those changes here.

D.2 Module Enhancements and Additions

During the development of Repo and Coterie, we created a wide range of new Repo modules, and enhanced a number of others. While we will not detail all of those changes here, we will highlight some of them to give an idea of the kinds of enhancements we made to

- *match* is the Repo null value `ok`
- *match* is *val*
- *val* is a text string and *match* is a regular expression that matches it exactly (i.e., *val* matches this regular expression: `"^" & match & "$"`)
- *match* is a regular expression that exactly matches the option key returned by `reflect_getType(val)` (i.e., *match* is *val*'s type)
- *match* is an option whose key is a regular expression that exactly matches the option key returned by `reflect_getType(val)`, and the value of the option is either `ok` or also matches *val*
- *val* and *match* are objects, and for each field of *match*, *val* has a corresponding field whose contents are matched by the contents of the field of *match*
- *val* and *match* are arrays of the same size, and each element of the array matches

Figure D-1: Pattern matching with the Repo reflection module. Pattern matching allows a programmer to create a prototype value *match* and check if the Repo value *val* matches it. *Match* will match a Repo value *val* if these condition are met.

the system, both large and small. See Appendix E for the details of all the built-in Repo modules, including the new and modified ones.

As mentioned in Appendix D.1, one of the modules we created, the `reflect` module, supports a simple form of *reflection* (see Appendix E.1.3). Reflection, as we implement it, is the ability for a programming language to operate on its type system. We implement this package primarily to allow programmers to check the types of parameter values, to make debugging large programs easier. However, the module goes beyond type checking. We can not only query values about their types (including generating option values with the tag strings describing the types), but operate extensively on objects. For example, the module supports invoking object methods, querying objects about the existence of fields and methods, extracting the fields of an object, and creating objects from those extracted fields.

One of the most useful functions in the `reflect` module is the `match` function, which supports object-based pattern matching, modeled after a similar facility in the Scheme Shell [Shivers, 1994]. The rules for constructing pattern matching values are shown in Figure D-1. Notice that if the match value is an object or array, the match process is performed recursively on the fields or elements of the match value. Therefore, arbi-

trarily complex structures can be matched, allowing fairly complex argument checking to be performed in a single step. We make use of the match facility in the Shared Sketch example in Section 6.2, to check that the objects imported from the network have all the required fields for use as Sketch objects. By doing this check, we can be confident that our code will not be broken either maliciously, or by bugs in other parts of the distributed system.

One final interesting change we made to the Obliq libraries was to modify the way filesystem and processor objects work. When a repo process starts, it has three variables defined in its environment (in this case, Repo is running on a host named `elvis`):

```
let processor = <Processor at elvis>
let fileSys = <FileSystem at elvis>
let fileSysReader = <FileSystem at elvis>
```

Filesystem objects are used to access the filesystem, and processor objects are used to create processes. `processor` is the local processor variable, `fileSys` is the local file system, and `fileSysReader` is a read-only version of the local file system. Since these handles are defined in the scope of the initial Repo thread, lexical scoping ensures that these handles can only be accessed by source code interpreted (as opposed to executed) by that thread. Therefore, these variables provide security to the local processor and file system. In Obliq, these objects cannot be transmitted across the network (doing so results in an exception). In Repo, we allow these variables to be transmitted over the network, where they always refer back to the resources in their original process. For example, we could have a group of processes transmit their processor variables to a single Repo process, which could then start processes on any of the machines containing those Repo processes.

D.3 Efficient Module Distribution

There is a subtle efficiency problem with the implementation of Obliq, related to the fact that the language is interpreted, that was not noticeable until we added replicated data to

the language. The problem is that the data structures representing modules (the primary code structuring mechanism in Obliq and Repo) can end up being transferred repeatedly over the network, and instantiated many times in remote processes. We will describe the problem, and the solution we adopted, because it illustrates the kind of subtle problem that can arise when building distributed applications. It also illustrates the importance of having a robust, general purpose infrastructure to free programmers from having to worry about such details.

To understand the problem (which will be described fully below), first consider what happens when an object is transferred between processes in Modula-3. To be successfully transferred, the object's type must exist in the remote process, which implies that the modules related to that object (that define the type, the methods, and so on) have been compiled into both programs. Therefore, when the object is transferred, a small identifier representing its type can be sent along with the instance data of the object.

Now, consider what happens in Repo (or Obliq). First, Repo objects are not typed, but are simply collections of fields, methods and aliases, so there is no simple way of identifying these objects between processes using a small identifier. Therefore, the entire object definition, including the closures defining the methods, and any free variables referenced from those methods, must be transmitted. This could represent a sizable amount of data, and there is no simple way to avoid it. This is not a serious problem in and of itself, as programmers will typically ensure that objects that are to be copied frequently do not have huge data structures embedded in them. Furthermore, objects that are copied frequently (such as events describing tracker or mouse motion) do not typically have huge numbers of methods.

However, if any of these methods reference a Repo module, a problem arises. Modules are the code structuring facility in Repo, and are used to group related procedures and data together. Unlike the compiled code in a Modula-3 binary (in which we can assume that similar programs are communicating), just because a set of modules has been loaded into a Repo process, we cannot assume they have been loaded into any other Repo process. Furthermore, even if modules with the same name, variables and procedures have

been loaded into two processes, there is no guarantee that they are actually the same module.

If we create an object that refers to some module (perhaps because the object's methods call procedures or reference variables in the module), and we pass that object between processes, we must ensure that the referenced module exists in the remote process. Therefore, the data structures defining the modules must be transferred along with the object. Furthermore, if the module refers to other modules, those must also be transferred at the same time. Since the internal module data structures are created with simple Modula-3 objects (i.e., unsynchronized replicated data), each time a reference to a module is sent to another process, the module definition is copied again.

This was not a serious problem in Obliq because copying happens infrequently. However, in Repo, serious network utilization and memory usage problems can occur as a result of these duplicate copies. Imagine that we are generating an unsynchronized replicated object each time a tracker moves, and distributing these objects. If this object references even a simple variable in some module, and that module happens to reference some other module, and so on, the resulting message could be huge (and take a significant amount of time to create and extract). Furthermore, the data structures representing these modules would be created repeatedly in the destination processes, potentially resulting in many copies of each module definition. This is clearly unacceptable.

There are a number of solutions to this problem. The most aggressive solution would be to recognize when two modules with the exact same code were loaded into different processes, and not copy module definitions that are not needed. However, because we operate in a heterogeneous environment, and modules can have virtually anything defined within them, an automated approach to this seemed difficult, and we did not want to resort to a manual approach (such as having programmers annotate the module source files with version numbers) because the chance of programmers accidentally introducing obscure bugs into their programs (by not changing the version numbers, for example) seemed high.

Instead, we adopted a more conservative, and significantly easier to implement, solution, ensuring that at most one copy of a particular module generated in a particular Repo process exists in any other Repo process. While multiple copies of some module could still exist in each process if the module was loaded into multiple processes and then transferred around, at least we limit the number of copies to a well defined number. The implementation assigns each module a unique network identifier (identifying both the module and the process in which it was generated) and passes that identifier around the network rather than the module definition. If a process receives an identifier for a module it does not currently have a copy of, it acquires the module from the process that created it. While distributing modules in this way is slightly less efficient when the module does not yet exist in the destination process (requiring an extra round trip on the network compared with sending the module data structures along with an object), it is significantly more efficient when the module data structures already exist in the destination process (which is the case we are worried about).

APPENDIX E Repo Modules

In Appendix D, we described a number of Repo modules that we created to enable building applications in our domain. In this appendix, we include the help files of all of the new and modified Repo modules to serve as a reference to the kinds of features we added to the system.

E.1 New Modules

E.1.1 debug

```
All(T) debug_assertFree (v: T)
   Assert that this value is free.
debug_checkHeap()
   Check the heap for all locations of any value that was
   asserted free.
debug_collectNow()
   A hint that this might be a good time to do a garbage collection
debug_reportReachable()
   Generate a report to stderr of all reachable data
debug_disableCollector()
   Prevent garbage collection
debug_enableCollector()
   All garbage collection to resume
debug_dumpReplicaState()
   Dump a report about the state of the replicated object runtime
debug_replicaDebugLevel(level: Int): Ok
   level>0 causes debugging info to go to stderr
```

The debug module exposes some of the Modula-3 debugging facilities in Repo. The last two routines are used for debugging the Shared Object runtime, and the rest are debugging routines implemented in the Modula-3 garbage collector.

E.1.2 dict

```
  dict_invalidKey: Exception
  dict_new (): Dict
    Create a new dictionary that maps Texts to any Obliq value.
  dict_get (t: Dict, key: Text): Val ! dict_invalidKey
    Look up "key" in dictionary "t".  If it exists, return the value
    "Val" that it maps to.  Otherwise, raise "dict_invalidKey".
  dict_put (t: Dict, key: Text, val: Val): Bool
    Set the value mapped from "key" in the dictionary "t" to "val".  If
    "key" already mapped to something in "t", return "true", otherwise
    return "false."
  dict_delete (t: Dict, key: Text): Val ! dict_invalidKey
    Delete the mapping for "key" from dictionary "t".  If it exists,
    return the value "Val" that it mapped to.  Otherwise, raise
    "dict_invalidKey".
  dict_size (t: Dict): Int
    Return the number of elements (mappings) in dictionary "t"
  dict_iterate (t: Dict): Iterator
    Return an Iterator, which is an object that can be used to iterate
    over the key-value pairs in "t".
  dict_iteratorNext (i: Iterator): [Text, Val] | Ok
    If "i" is the result of the call "dict_iterate(t)", then the call
    "dict_iteratorNext(i)" selects an entry from "t" that has not
    already been returned by "i", and returns the pair ["k","v"]
    corresponding to its key and value.  If no entries remain, the call
    returns "Ok". It is a checked runtime error to call "iteratorNext"
    after it has returned "Ok". The client must ensure that while an
    iterator is in use, the parent dictionary is not modified.
  dict_iteratorInit (i: Iterator, t: Dict): Iterator
    Reinitialize "i" to iterate over all the values of a dictionary "t".
    Return "i"
WHERE
  Dict is a dictionary
  Iterator is a dictionary iterator
```

This module exposes a dictionary (implemented using a hash table) into Repo, which maps text keys to any Repo value. We decided to implement this as a module, rather than extending the language to support associative arrays.

E.1.3 reflect

```
  reflect_error: Exception
  All(T) reflect_isArray(v: T): Bool
    is v an array?
  All(T) reflect_isObject(v: T): Bool
    is v an object?
  All(T) reflect_isClosure(v: T): Bool
    is v a closure?
  All(T) reflect_isException(v: T): Bool
```

```
  is v an exception?
All(T) reflect_isMethod(v: T): Bool
  is v a method?
All(T) reflect_isUpdateMethod(v: T): Bool
  is v an update method?
All(T) reflect_isOption(v: T): Bool
  is v an option?
All(T) reflect_isBasic(v: T): Bool
  is v a basic value? (ok, Bool, Char, Text, Int, Real)
All(T) reflect_isNative(v: T): Bool
  is v a native value? (an opaque value)
All(T) reflect_isAlias(v: T): Bool
  is v an alias?
All(T) reflect_isLocal(v: T): Bool
  is the location of v local to this site?
All(T) reflect_isProtected(v: T): Bool
  is v a protected object?
All(T) reflect_isSerialized(v: T): Bool
  is v a serialized object?
All(T) reflect_isSimple(v: T): Bool
  is v a simple object?
All(T) reflect_isReplicated(v: T): Bool
  is v a replicated object?
All(T) reflect_isRemote(v: T): Bool
  is v a remote object?
All(T<:option(tag,val)) reflect_getOptionTag(o: T): tag
  return the tag of the option
All(T<:option(tag,val)) reflect_getOptionVal(o: T): val
  return the value of the option
All(T) reflect_getType(v: T): option(type,ok)
  return an option whose label describes the type of v, and whose
  value is ok
All(T) reflect_getTypedVal(v: T): option(type,v)
  return an option whose label describes the type of v, and whose
  value is v
All(T<:{}) reflect_getFieldTypes(v: T): [[Text,option(type,ok)]]
  return an array describing the fields of v.  Each array element is a
  2 element array containing the field label and an option describing
  its type. The value of the option is always ok.
All(T<:{}) reflect_getObjectType(v: T): option(objectType,v)
  return an option created by appending all the option tags of the
  fields of v using the text template "label=>tag" for each field
  Obtaining the type of a remote object does not require the field
  values to be copied to the local machine.
All(T<:{}) reflect_getObjectInterface(v: T): option(objectType,v)
  similar to getObjectType, but only methods of v are included.
All(T<:{}) reflect_objectWho(v: T): Text
  return the text that is used to identify the object v when it is
  printed
All(T<:{}) reflect_getField(v: T, label: Text) ! reflect_error
  get the named field from the object.
All(T<:{}) All(S) reflect_getFields(v: T): [[Text,S]] ! reflect_error
  return an array of pairs of field labels and their values.
All(T<:{}) reflect_select(v: T, label: Text): S ! reflect_error
  the same as calling 'v.label'.
All(T<:{}) All(S,U) reflect_update(v: T, label: Text, nv:S):U !
            reflect_error
  the same as calling 'v.label := nv'.
All(T<:{}) All(S,U) reflect_invoke(v: T, label: Text,
```

```
                                    args: [S]): U !reflect_error
    the same as calling 'v.label(args)', where args is expanded to an
    arg list
  All(T<:{}) All(S) reflect_newObject(v: ObjectType,
                        protected serialized: Bool,
                        who: Text, fields: [[Text,val]]): S
    create a new object.
  All(T,U) reflect_match(match: T, val: S): bool !reflect_error
    Test to see if "val" matches "match", using the rules below.
WHERE
  ObjectType = one of {"Remote","Replicated","Simple"}

  A value "val" matches a "match" value if:
  - "match is ok"
  - "match is val"
  - "val" is a text string and "match" is a regular expression that
    matches all of it (ie. "val" matches "^" & match & "$")
  - "match" is a regular expression that matches all of the option key
    of "reflect_getType(val)"
  - "match" is an option whose key is a regular expression matches all
    of the option key of "reflect_getType(val)", and the value of the
    option is either "ok" or also matches "val".
  - "val" and "match" are objects, and for each field of "match",
    "val" has a corresponding field whose contents are matched by the
    contents of the field of "match".
  - "val" and "match" are arrays of the same size, and each element of
    the array matches

  Here are the possible types strings of the basic Repo types:
  "Var", "Var`Replicated", "Var`Simple", "Ok", "Bool", "Char", "Text",
  "Int", "Real", "Option", "Alias", "Array`Remote", "Array`Replicated",
  "Array`Simple", "Closure`#", "Method`#`Update", "Method`#",
  "Object`Remote", "Object`Replicated", "Object`Simple", "Engine",
  "Exception"

  Opaque data types introduced by libraries have a type string of either
  "ValAnything" or a value provided by the library.
```

This module adds reflection to Repo, as discussed in Appendix D.

E.1.4 replica

```
  replica_failure: Exception
  replica_fatal: Exception
  All(T<:[replica]{}, S:[simple]{}) replica_notify(o: T, n: S): callback
    ! replica_failure
  All(T<:[replica]{}) replica_cancelNotifier(cb: callback)
  replica_flushIncomingUpdates(): Ok ! thread_alerted
  replica_flushQueuedUpdates(): Ok ! thread_alerted
```

This module defines the replicated object exceptions, and provides the functions to create and destroy Shared Object callbacks in Repo. The callback is a simple object with meth-

ods corresponding to the pre and post updates that the programmer wishes to be informed of. The module also exposes the Shared Object runtime routines to flush the update queue, as described in Section 3.4.2.1.

E.2 New Modules for Modula-3 Packages

The modules in this section are new to Repo, but simply expose existing Modula-3 packages that we needed access to in Repo.

E.2.1 dir

```
dir_failure: Exception
dir_getAbsolutePathname(fs: FileSystem, p: Text): Text ! dir_failure
  Return an absolute pathname referring to the same file or
  directory as "p".  The new pathname will not involve any symbolic
  links or relative arcs (that is, occurrences of "path_parent" or
  "path_current".
dir_createDirectory(fs: FileSystem, p: Text): Ok ! dir_failure
  Create a directory named by "p".
dir_deleteDirectory(fs: FileSystem, p: Text): Ok ! dir_failure
  Delete the directory named by "p".  "dir_failure" is raised if the
  directory contains entries (other than perhaps "path_current"
  and "path_parent").
dir_deleteFile(fs: FileSystem, p: Text): Ok ! dir_failure
  Delete the file or device named by "p".  "dir_failure" is raised if
  "p" names a directory.
  Note: Under Win32, "DeleteFile" raises "dir_failure" if "p" is open.
  Under POSIX, an open file may be deleted; the file doesn't actually
  disappear until every link (path) for it is deleted.
dir_rename(fs: FileSystem, p0 p1: Text): Ok ! dir_failure
  Rename the file or directory named "p0" as "p1".
  Some implementations automatically delete an existing file named
  "p1", others raise "dir_failure".  Some implementations disallow a
  rename where "p0" and "p1" name different physical storage devices
  (different root directories or file systems).
dir_iterate(fs: FileSystem, p: Text): Iterator ! dir_failure
  Return an iterator for the entries of the directory named by "p".
  An "Iterator" supplies information about the entries in a
  directory: names and, optionally, status.  The iteration does not
  include entries corresponding to "path_current" or "path_parent".
dir_iteratorNext(i: Iterator): [Text, Bool]
  If more entries remain, returns ["n",True], with "n" set to the name
  of the next one.  It returns ["n",False], with "n"  undefined, if no
  more entries remain.
dir_iteratorNextWithStatus(i: Iterator): [Text, Bool, Status] !
          dir_failure
  If more entries remain, returns ["n",True,Status], with "n" set to
  the name of the next one and "Status" set to its status (see
  dir_status). It returns ["n",False,ok], with "n" undefined, if no
```

```
    more entries remain.
  dir_iteratorClose(i: Iterator): Ok
    The call "i.close()" releases the resources used by "i", after
    which time it is a checked runtime error to use "i".  Every
    iterator should be closed.
  dir_status(fs: FileSystem, p: Text): Status ! dir_failure
    Return information about the file or directory named by "p".
    The type field includes the values "Directory" for directories,
    "RegularFile" for disk files,"Terminal" for terminals and "Pipe" for
    pipes.
  dir_setModificationTime(fs: FileSystem, p: Text, t: Real): Ok !
          dir_failure
    Change the modification time of the file or directory named by "p"
    to "t".
WHERE
  Iterator is a directory iterator
  FileSystem is a file system.  The local file system is available
    through the predefined lexically scoped identifier "fileSys".
  Status = {type => Text, modificationTime => Real, size => Int};
```

This module exposes the Modula-3 directory manipulation routines into Repo. Paths are specified in a OS independent fashion using the `path` module (Section E.2.5).

E.2.2 http

```
  http_error: Exception
  http_notAuthorized: Exception
  http_badQuery: Exception

  http_logging(on: Bool): Ok
    turn logging on and off
  http_setDefaultViaFieldValue(v: Version, port: Int, alias: Text): Ok
    generate and set the default viaFieldValue for the default style
    This field MUST be set for proxies.  If alias is not "", it is used
    in place of the host name.
  http_toText(h: Header, proxy: Bool): Text ! http_error
  http_lookupField(h: Header, name value: TEXT): Field
  http_addField(h: Header, field after: Field): Field
  http_removeField(h: Header, field: Field): Bool
  http_copyFields(from to: Header)
  http_iterateFields(h: Header): FieldIterator
  http_iterateNextField(i: FieldIterator): Field

  http_newRequest(m: Method, url: URL, v: Version): Request
    create a new request
  http_parseRequest(rd: Rd): Request ! http_error
    parse a request header from rd
  http_writeRequest(r: Request, wr: Wr, proxyRequest: Bool) ! http_error
  http_requestMethod(r: Request): Method
  http_requestURL(r: Request): URL
  http_requestVersion(r: Request): Version
  http_requestPostData(r: Request): Text
```

```
http_version9: Version
http_version10: Version
http_version11: Version
http_currentVersion: Version
  supported HTTP versions (0.9, 1.0, 1.1)

http_statusCode(status: HttpStatus): Int
http_statusReason(status: HttpStatus): Text

http_newReply(v: Version, code: Int, reason: Text): Reply
  create a new reply
http_parseReply(rd: Rd): Reply ! http_error
  parse a reply header from rd
http_writeReply(r: Reply, wr: Wr): Ok ! http_error
http_replyVersion(r: Reply): Version
http_replyCode(r: Reply): Int
http_replyReason(r: Reply): Text
http_writeSimpleReplyHeader(wr: Wr, code: Int, reason: Text): Ok !
        http_error
http_writeRedirectReply(wr: Wr, url, htmlMsg: Text): Ok ! http_error

http_writeTime(wr: Wr, time: Real): Ok ! http_error
http_readTime(rd: Rd): Real ! http_error
http_setProgramInfo(prog: ProgramType, name: Text, auth: AuthType,
                    authRealm authAccount: Text): Ok
http_getProgramInfo(): {programType => ProgramType, name => Text,
              authType => AuthType, authRealm => Text,
              authAccount => Text}

http_newFormQuery(query: Text): FormQuery ! http_badQuery
http_newFormQueryFromRd(rd: Rd): FormQuery ! http_badQuery
  parse a query from a text or rd
http_writeFormQuery(f: FormQuery, wr: Wr): Ok ! http_error

http_basicAuthField(account: Text, auth: AuthType): Field
  create a Basic authorization field where account is "name:passwd"
http_authorizedRequest(r: Request, auth: AuthType,
                       account: Text): Bool ! http_error
  check if request has a valid auth field for account
http_replyUnauthorized(wr: Wr, auth: AuthType, realm: Text,
  defaultMsg: Bool): Ok ! http_error
  write an "unauthorized" reply to wr for realm.  If "defaultMsg",
  write a simple message.
http_authorizationAccount(r: Request, auth: AuthType): Text !
        http_error
  return the authorization field
http_readBody(h: Header, rd: Rd, dest:(data:text)->Ok): Ok !
        http_error
  read the body from "rd" by calling "dest" as necessary
http_writeBody(h: Header, wr: Wr, src:(len:Int)->Text): Ok !
        http_error
  write the body to "wr" by calling "src" as necessary.  the end of
  the body is signified by "src" returning less than "len" characters

http_escapeURLEntry(entry: Text): Text
http_unescapeURLEntry(entry: Text): Text ! http_error
http_encodeTextForHTML(text: Text): Text
http_decodeTextForHTML(text: Text): Text ! http_error
```

```
http_getUserAgent(r: Request): [Text,Int]
  return the agent name and version number

http_addProxy(rule: Text): Ok
  add a rule to the proxy server list.
  Rules are of the form "pattern <server>,[<server>]*"

http_anyPort: Int
http_anyService: Int
All(T) http_serve(port,service: Int, serverData: T): Ok ! http_error
  enter wait loop for HTTP requests on "port". "serverData" is passed
  to the "accept" and "request" methods of the RequestHandlers
http_serverPort(port,service: Int): Bool
  return True if there has been a call on "serve" for "port"
http_client(r: Request, v: Version, rd: Rd, wr: Wr,
            hander: (Reply,rd,wr)->Ok, service): Ok ! http_error
  Make a client request or proxy a client request.  The request is
  made directly if the destination server does not match against the
  noProxy list.  Program information (user-agent, or via) and host
  header is added automatically to the request. The contents of "rd"
  are sent with the request. After "request" is sent to the server,
  the header of the reply is parsed and "handler" is called with "wr"
  for its output.
All(S,T) http_registerRequestHandler(port: Int, pr: Priority,
            accept: (Request,S)->[T,Bool],
            request: (Request,S,T,Rd,Wr)->Ok): Ok
  Register a server request handler for a port. For an incoming
  request, all handler "accept" procedures are called (in Priority
  order) until one returns True.  The corresponding "request"
  procedure is then called. S is the "serverData" item passed to
  http_serve.  The "T" returned by "accept" is passed to "request".
  The handler will only get called if "port" matches the server's port
  or "port = AnyPort" or if "port" < 0 then "port" represents a
  service type, and the request handler is invoked if "port" matches
  the server's service type.
http_serverPushSupported(r: Request): Bool
http_serverPushFrame(wr: Wr, contentType, msg: Text): Ok ! http_error

http_rootForm: Form
  Forms and Values provide an interface for applictions to be
  controlled via an HTTP form interface.  This returns the root
  control form.
All(T) http_newForm(name: Text, accept: (Form,Request,Text)->[T,Bool],
                  respond: (Form,Request,FormQuery,Wr,T)->Ok !
                                 http_notAuthorized): Ok
  A specialized request handler for forms.
http_formName(f: Form): Text
http_iterateValues(f: Form): ValueIterator
http_iterateNextValue(i: ValueIterator): Value
  Iterate the values of a form.
http_registerForm(f: Form, name url: Text, addToRoot: Bool): Ok
  Register the form so that the form's accept procedure is called to
  see if the form handles the request.  If "addToRoot" the form
  is added to the root form.
http_formLookup(name: Text): Form
  returns the form registered under "name", or ok if there is no form
  registered under that name.
http_newStaticForm(name url title: Text, hasButton register: Bool):
         StaticForm
```

```
  A StaticForm is a form that has a fixed URL for its address and
  fixed contents (made up of values).
http_staticFormUrl(f: StaticForm): Text
http_staticFormAddValue(f: StaticForm, v: Value): Value

http_newValue(id: Text, [leader label trailer]: [Text],
       editable: Bool,
       getText: (self,Request)->Text  ! http_notAuthorized,
       setText: (self,Request,Text)->Ok ! http_notAuthorized,
       setDefault: (self,Request)->Ok ! http_notAuthorized,
       writeFormItem: (self,Request,Wr)->Ok ! http_notAuthorized):
            Value
  a generic form value. getText and setText retrieve and set the text
  representation of the value contents.  setDefault restores the value
  to its default state.  writeFormItem writes the html form contents
  to wr.
http_newContainerValue(id: Text, [leader label trailer]: [Text],
     editable: Bool,
     getText: (self,Request)->Text ! http_notAuthorized,
     setText: (self,Request,Text)->Ok ! http_notAuthorized,
     setDefault: (self,Request)->Ok ! http_notAuthorized,
     writeFormItem: (self,Request,Wr)->Ok ! http_notAuthorized,
     setValues: (self,Request,FormQuery)->Ok ! http_notAuthorized):
       ContainerValue
  a generic form container value. getText and setText retrieve and set
  the text representation of the value contents.  setDefault restores
  the value to its default state.  writeFormItem writes the html form
  contents to wr.  setValues sets the container values from an HTTP
  form query.
http_valueId(v: Value): Text
http_valueLeader(v: Value): Text
http_valueLabel(v: Value): Text
http_valueTrailer(v: Value): Text
http_valueEditable(v: Value): Boolean
http_setValueId(v: Value, id: Text): Text
http_setValueLeader(v: Value, leader: Text): Text
http_setValueLabel(v: Value, label: Text): Text
http_setValueTrailer(v: Value, trailer: Text): Text
http_setValueEditable(v: Value, editable: Bool): Bool
  retrieve and set value attributes.  The set functions return
  their arguments.
http_valueText(v: Value,r: Request): Text ! http_notAuthorized
http_setValueText(v: Value, r: Request,
                txt: Text): Ok ! http_error http_notAuthorized
  Most kinds of values get be set from a text representation of their
  value, which is how the forms are set from an HTTP POST.
http_valueSetDefault(v: Value, r: Request): Ok! http_error
       http_notAuthorized
  reset the value to its default.
http_writeFormItem(v: Value, r: Request,
                  wr: Wr): Ok ! http_error http_notAuthorized
http_setContainerValues(v: ContainerValue, r: Request,
                  q: FormQuery): Ok ! http_error http_notAuthorized
  Set the subvalues of a container value from an HTTP query

http_newBooleanValue(id: Text, [leader label trailer]: [Text],
            editable: Bool,
            get: (self,Request)->Bool ! http_notAuthorized,
            set: (self,Request,Bool)->Ok ! http_notAuthorized): Value
```

```
  http_newIntValue(id: Text, [leader label trailer]: [Text],
              editable: Bool,
              get: (self,Request)->Int ! http_notAuthorized,
              set: (self,Request,Int)->Ok ! http_notAuthorized): Value
  http_newRealValue(id: Text, [leader label trailer]: [Text],
              editable: Bool,
              get: (self,Request)->Real ! http_notAuthorized,
              set: (self,Request,Real)->Ok ! http_notAuthorized): Value
  http_newImageValue(id: Text, [leader label trailer]: [Text],
              get: (self,Request)->URL ! http_notAuthorized,
              set: (self,Request,URL)->Ok ! http_notAuthorized): Value
    an image
  http_newUrlValue(id: Text, [leader label trailer]: [Text],
              get: (self,Request)->Text ! http_notAuthorized,
              set: (self,Request,Text)->Ok ! http_notAuthorized): Value
    a link
  http_newFormValue(id: Text, [leader label trailer]: [Text],
              f: Form, name url: Text): Value
  http_newMsgValue(leader trailer msg: Text): Value
    a non-editable message
  http_newChoiceValue(id: Text, [leader label trailer]: [Text],
              editable: Bool,
              names: [Text],
              get: (self,Request)->Int ! http_notAuthorized,
              set: (self,Request,Int)->Ok ! http_notAuthorized): Value
    a list of alternatives
  http_newTextValue(id: Text, [leader label trailer]: [Text],
              editable: Bool,
              scrollable: Bool, width height: Int,
              get: (self,Request)->Text ! http_notAuthorized,
              set: (self,Request,Text)->Ok ! http_notAuthorized): Value
    a text area.  It is may or may not be scrollable.
  http_textValueDim(v: Value): [Bool,Int,Int] ! http_error
  http_setTextValueDim(v: Value, scrollable: Bool,
                       width height: Int): Ok! http_error
    change the properties of the text area
  http_newTableValue(id: Text, caption: Text,
                     values:[[Value]]): ContainerValue
  http_tableValue(request: Request,
                  v: Value): [[Value]] ! http_error http_notAuthorized
  http_setTable(request: Request, v: Value,
    values: [[Value]]): Ok ! http_error http_notAuthorized
    a table is a specific container value that creates an HTML Table.
WHERE
  Request <: Header
  Reply <: Header
  FormQuery <: Header
  ContainerValue <: Value
  Method = Text(one of "OPTIONS", "GET", "POST", "PUT",
                       "DELETE", "HEAD", "TRACE", "CONNECT")
  ProgramType = Text    (one of "Client", "Proxy", "Server", "Tunnel")
  AuthType = Text       (one of "None", "Proxy", "Server")
  Priority = Text       (one of "High", "Normal", "Low")
  Field = [Text,FieldValue]
  FieldValue = Text or ok
  Iterator = FieldIterator or ValueIterator
  FieldOrValue = Field or Value
```

This module exposes the Modula-3 HTTP package into Repo. It supports the creation of HTTP clients and servers, and includes support for authentication, getting and putting data to/from the servers, proxying and forms. The forms facility includes support for both simple predefined forms, and more general forms creation. The HTTP package was originally created as part of the Millicent project in electronic microcommerce at DEC SRC [Manasse, 1995].

E.2.3 httpField

```
httpField_accept: Text
httpField_acceptCharset: Text
httpField_acceptEncoding: Text
httpField_acceptLanguage: Text
httpField_acceptRanges: Text
httpField_age: Text
httpField_allow: Text
httpField_authorization: Text
httpField_cacheControl: Text
httpField_connection: Text
httpField_contentBase: Text
httpField_contentEncoding: Text
httpField_contentLanguage: Text
httpField_contentLength: Text
httpField_contentLocation: Text
httpField_contentMD5: Text
httpField_contentRange: Text
httpField_contentType: Text
httpField_date: Text
httpField_eTag: Text
httpField_expires: Text
httpField_from: Text
httpField_host: Text
httpField_ifModifiedSince: Text
httpField_ifMatch: Text
httpField_ifNoneMatch: Text
httpField_ifRange: Text
httpField_ifUnmodifiedSince: Text
httpField_lastModified: Text
httpField_location: Text
httpField_maxForwards: Text
httpField_pragma: Text
httpField_proxyAuthenticate: Text
httpField_proxyAuthorization: Text
httpField_public: Text
httpField_range: Text
httpField_referer: Text
httpField_retryAfter: Text
httpField_server: Text
httpField_transferEncoding: Text
httpField_upgrade: Text
```

```
  httpField_userAgent: Text
  httpField_vary: Text
  httpField_via: Text
  httpField_warning: Text
  httpField_WWWAuthenticate: Text
    Return the text value of the field name.  This interface is provided
    to ensure only valid field names are used unless explicitely
    intended.
```

The package is used by the HTTP package (Section E.2.2). It predefines all of the HTTP protocol field names.

E.2.4 httpStatus

```
  httpStatus_continue: HttpStatus
  httpStatus_switchingProtocols: HttpStatus
  httpStatus_ok: HttpStatus
  httpStatus_created: HttpStatus
  httpStatus_accepted: HttpStatus
  httpStatus_nonAuthoritative_Information: HttpStatus
  httpStatus_noContent: HttpStatus
  httpStatus_resetContent: HttpStatus
  httpStatus_partialContent: HttpStatus
  httpStatus_multipleChoices: HttpStatus
  httpStatus_movedPermanently: HttpStatus
  httpStatus_movedTemporarily: HttpStatus
  httpStatus_seeOther: HttpStatus
  httpStatus_notModified: HttpStatus
  httpStatus_useProxy: HttpStatus
  httpStatus_badRequest: HttpStatus
  httpStatus_unauthorized: HttpStatus
  httpStatus_paymentRequired: HttpStatus
  httpStatus_forbidden: HttpStatus
  httpStatus_notFound: HttpStatus
  httpStatus_methodNotAllowed: HttpStatus
  httpStatus_notAcceptable: HttpStatus
  httpStatus_proxyAuthenticationRequired: HttpStatus
  httpStatus_requestTimeout: HttpStatus
  httpStatus_conflict: HttpStatus
  httpStatus_gone: HttpStatus
  httpStatus_lengthRequired: HttpStatus
  httpStatus_preconditionFailed: HttpStatus
  httpStatus_requestEntityTooLarge: HttpStatus
  httpStatus_requestURITooLarge: HttpStatus
  httpStatus_unsupportedMediaType: HttpStatus
  httpStatus_internalServerError: HttpStatus
  httpStatus_notImplemented: HttpStatus
  httpStatus_badGateway: HttpStatus
  httpStatus_serviceUnavailable: HttpStatus
  httpStatus_gatewayTimeout: HttpStatus
  httpStatus_httpVersionNotSupported: HttpStatus
WHERE
  HttpStatus is a predefined HTTP Status code.  The corresponding code
```

```
and textual reason can be obtained from the http interface.
```

The package is used by the HTTP package (Section E.2.2). It predefines all of the HTTP protocol status codes.

E.2.5 path

```
path_invalid: Exception
  When a path with invalid syntax is passed to a procedure in
  this interface not declared as raising the exception "invalid",
  the result is undefined, but safe.
path_valid(fs: FileSystem, pn: Text): Bool
  Return "True" iff "pn" conforms to the path syntax of this
  operating system.
path_decompose(fs: FileSystem, pn: Text): [Text] ! path_invalid
  Parse "pn", returning a sequence whose first element is a root
  directory name (possibly "") and whose remaining elements
  consist of zero or more arc names.  Raise "path_invalid" if
  "path_valid(pn)" is "False". "path_decompose" returns exactly the
  sequence of arc names present in "pn"; it doesn't attempt to produce
  a canonical form. Some operating systems allow zero-length arc
  names.
path_compose(fs: FileSystem, arcs: [Text]): Text ! path_invalid
  Combine the elements of "arcs" to form a path corresponding to the
  syntax of this operating system.  Raise "path_invalid" if "arcs" is
  [], if "arcs[0]" is neither "" nor a valid root directory name, or
  if one of the elments of "arcs" is not a valid arc name.
path_absolute(fs: FileSystem, pn: Text): Bool
  Return "True" iff "pn" is an absolute path.  Equivalent to
  "not(text_equal(path_decompose(pn)[0],""))", but faster.
path_prefix(fs: FileSystem, pn: Text): Text
  Return a path equal to "pn" up to, but not including, the final
  arc name.  If "pn" consists only of a root directory name,
  "path_prefix(pn)" returns "pn".
path_last(fs: FileSystem, pn: Text): Text
  Return the final arc name in "pn".  If "pn" consists only of a root
  directory name, "path_last(pn)" returns the empty string.
path_base(fs: FileSystem, pn: Text): Text
  Return a path equal to "pn" except with "path_last(pn)" replaced by
  its base.
path_join(fs: FileSystem, pn base ext: Text): Text
  Return a path formed by prepending "pn" to "base" (if "pn" is
  not "") and appending "ext" to "base" (if "ext" is not "").
  More precisely, this is equivalent to the following, in which "a"
  is an array of Text:
    if text_equal(pn, "") then a := [];
    else
      if path_absolute(base) then `Cause checked runtime error` end;
      a := path_decompose(pn);
    end;
    if text_length(ext) > 0 then base := base & "." & ext end;
    let ba = path_decompose(base);
  path_compose(a @ ba[1 for (#(ba)-1)]);
```

```
    The value returned by "path_join" will be a valid path only if the
    "base" and "ext" conform to the syntax of the particular operating
    system.
  path_lastBase(fs: FileSystem, pn: Text): Text
    Return the base of the final arc name of "pn".  It is a checked
    runtime error if "pn" is empty or consists only of a root directory
    name.
  path_lastExt(fs: FileSystem, pn: Text): Text
    Return the extension of the last arc name of "pn".  It is a checked
    runtime error if "pn" is empty or consists only of a root directory
    name.
  path_replaceExt(fs: FileSystem, pn ext: Text): Text
    Return a path equal to "pn" except with the extension of the
    final arc name replaced with "ext", which must not be "".
  path_parent(fs: FileSystem): Text
    A special arc name that, when encountered during a path lookup,
    stands for the parent of the directory currently being examined.
  path_current(fs: FileSystem): Text
    A special arc name that, when encountered during a path lookup,
    stands for the directory currently being examined.
  path_searchSeparator(fs: FileSystem): Text
    The search path separator charactor, used for appending multiple
    paths together.
  path_separator(fs: FileSystem): Text
    The path separator charactor.  Used to separate the arcs in a path.
```

This module exposes the Modula-3 path manipulation routines into Repo. The path module is used to manipulate pathnames in an operating system independent way.

E.2.6 random

```
  random_int(min, max: Int): Int
  random_real(min, max: Real): Real
    return a random number in the range [min, max]
```

The module exposes the Modula-3 random number generator into Repo.

E.2.7 regex

```
  regex_error: Exception
  regex_compile(pat: Text): Pattern ! regex_error
    compile a regular expression string into an regular expression
    Pattern
  regex_decompile(pat: Pattern): Text
    decompile an executable Pattern into the original regular expression
  regex_dump(pat: Pattern): Text
    dump an executable Pattern into a readable text string for debugging
  regex_execute(pat: Pattern, text: Text): Int
```

```
  compare the regular expression 'pat' against the text data returning
  the starting position in 'data' if there was a match, -1 otherwise.
regex_executeRes(pat: Pattern, text: Text): [[Int,Int]] or ok
  if there is a match, return indices for the (..) sequences.
  Otherwise, return ok.
regex_executeSub(pat: Pattern, text: Text, start len: Int): Int
regex_executeSubRes(pat:Pattern, text:Text,
                    start len:Int): [[Int,Int]] or ok
  consider only text_sub(text,start,len) portion of 'text'
```

This module exposes the regular expression package into Repo. This is the regex package that is also used to implement the regular expression matching in the Repo `case` statement, and in the `reflect` module.

E.2.8 tcp

```
tcp_error: Exception
tcp_getHostByName(name: Text): Ok or Address ! tcp_error
  look up the IP address of a host.  Return Ok if the host cannot be
  found
tcp_getCanonicalByName(name: Text): Text ! tcp_error
tcp_getCanonicalByAddr(addr: Address): Text ! tcp_error
  return the canonical host name
tcp_getHostAddr(): Address
  return one of this hosts address
tcp_newConnector(ep: Endpoint): Connector ! tcp_error
  the address portion should be zeros or a valid IP address of this
  host. if the port is zero, a free one will be chosen.  Use
  getEndpoint to find out which one
tcp_getEndPoint(conn: Connector): Endpoint
  get the endpoint of the Connector
tcp_closeConnector(conn: Connector)
  close the Connector
tcp_connect(ep: Endpoint): T ! tcp_error thread_alerted
  connect to some TCP address
tcp_accept(conn: Connector): T ! tcp_error thread_alerted
  accept an incoming connection on a Connector
tcp_close(tcp: T)
  close a TCP connection
tcp_eof(tcp: T)
  returns "True" if and only if there are no more bytes to be read
  from this connection, and the connection indicates end-of-file (e.g.
  the other side closed it.
tcp_startConnect(ep: Endpoint): T ! tcp_error
  initiate a request to connect to the destination specified by "ep".
tcp_finishConnect(tcp: T, waitFor: Real): Bool ! tcp_error
            thread_alerted
  returns a "Bool" to indicate if a connection request initiated via
  "startConnect" has successfully completed. A result of "True"
  indicates that it has.  "False" means that the connection request is
  still outstanding. If "waitFor" is negative, then "finishConnect"
  waits indefinitely until the operation completes, otherwise it waits
```

```
    for a maximum of "waitFor" seconds.  The caller should continue to
    call this procedure until it either returns "True" or raises an
    error.
  tcp_getPeer(tcp: T): Endpoint ! tcp_error
    return the peer endpoint for TCP connection.
  tcp_getPeerName(tcp: T): Text ! tcp_error
    return the peer name for TCP connection.
  tcp_matchPeer(tcp: T, addr: Address, maskBits: MaskBits): Bool !
              tcp_error
    returns "True" if the first maskBits bits of peer's endpoint address
    match the given address.
  tcp_localEndpoint(tcp: T): Endpoint ! tcp_error
    return the local Endpoint of a TCP connection.
  tcp_getRd(tcp: T): Rd
    get a reader on the TCP connection
  tcp_getWr(tcp: T): Wr
    get a writer on the TCP connection
WHERE
  MastBits = [0 .. 32]
  Address = [Int,Int,Int,Int]
    A valid IP address
  Endpoint = [Int,Int,Int,Int,Int]
    A valid IP address and port number
  Connector = an opaque TCP connector
```

In this module, we expose the Modula-3 TCP and IP modules. This allows simple TCP-based communication to be implemented at the Repo level, which is needed to communicate with other, non-Modula-3 programs. We used this module in the Shared Sketch example, in Section 6.2, to communicate with the Brown Sketch system.

E.2.9 url

```
  url_new(textRep: Text): URL
  url_newFromRd(rd: Rd): URL
  url_toText(url: URL, f: Format): Text
  url_equivalent(url1,url2: URL): Bool
  url_local(url: URL, service: Int): Bool
  url_derelativize(self, root: URL): URL
  url_absPath(url: URL): Bool
  url_scheme(url: URL): Text
  url_host(url: URL): Text
  url_port(url: URL): Int
  url_path(url: URL): Text
  url_params(url: URL): Text
  url_query(url: URL): Text
  url_fragment(url: URL): Text
WHERE
  Format = Text(one of "Default", "Canonical", "BodyOnly")
```

This module is used with the HTTP module (Section E.2.2) to provide a high-level way of manipulating URLs.

E.2.10 word

```
word_bitnot(w: Int): Int
  the bitwise not of w.
word_bitand(w1 w2: Int): Int
  the bitwise and of w1 and w2.
word_bitor(w1 w2: Int): Int
  the bitwise or of w1 and w2.
word_bitxor(w1 w2: Int): Int
  the bitwise xor of w1 and w2.
word_bitshift(w n: Int): Int
  the bitwise shift of w by n bits.
word_bitrotate(w n: Int): Int
  the bitwise rotate of w by n bits.
```

The word module exposes the Modula-3 bitwise word manipulation operators.

E.3 Changed Modules

The modules in this section existed in Obliq, but were enhanced in (sometimes significant) ways in Repo.

E.3.1 array

```
[e1, ..., en]: [T]
  (for e1...en: T).  Creates a remote array.
All(T) array_new(size: Int, init: T): [T]
All(T) array_newRemote(size: Int, init: T): [T]
All(T) array_newReplicated(size: Int, init: T): [T]
All(T) array_newSimple(size: Int, init: T): [T]
  A remote, replicated or simple array of size 'size', all filled
  with 'init'.  'new' is a shorthand for 'newRemote'.
All(T) array_gen(size: Int, proc: (Int)->T): [T]
All(T) array_genRemote(size: Int, proc: (Int)->T): [T]
All(T) array_genReplicated(size: Int, proc: (Int)->T): [T]
All(T) array_genSimple(size: Int, proc: (Int)->T): [T]
  A remote, replicated or simple array of size 'size', filled with
  'proc(i)' for 'i' between '0' and 'size-1'.  'new' is a shorthand
  for 'genRemote'
All(T) array_#(a: [T]): Int ! net_failure
  (also '#(a)') Size of an array.
All(T) array_get(a: [T], i: Int): T ! net_failure
```

```
   (also 'a[i]') The i-th element (if it exists), zero-based.
All(T) array_set(a: [T], i: Int, b: T): Ok ! net_failure
   (also 'a[i]:=b') Update the i-th element (if it exists).
All(T) array_sub(a: [T], i: Int, n: Int): [T] ! net_failure
   (also 'a[i for n]') A new array, of the same kind as 'a', filled
   with the elements of 'a' beginning at 'i', and of size 'n' (if it
   exists).
All(T) array_upd(a: [T], i: Int, n: Int, b: [T]): Ok ! net_failure
   (also 'a[i for n]:=b') Same as 'a[n+i]:=b[n]; ... ; a[i]:=b[0]'.
   I.e. 'a[i for n]' gets 'b[0 for n]'.
All(T) array_@(a1: [T], a2: [T]): [T] ! net_failure
   (also infix '@') A new array, of the same kind as 'a1', filled with
   the concatenation of the elements of 'a1' and 'a2'.
```

A set of new constructor functions was added to this module to create arrays with different distribution semantics.

E.3.2 fmt

```
fmt_padLft(t: Text, length: Int): Text
   If t is shorted then length, pad t with blanks on the left so that
   it has the given length.
fmt_padRht(t: Text, length: Int): Text
   If t is shorted then length, pad t with blanks on the right so that
   it has the given length.
fmt_bool(b: Bool): Text
   Convert a boolean to its printable form.
fmt_int(n: Int): Text
   Convert an integer to its printable form.
fmt_real(r: Real): Text
   Convert a real to its printable form.
fmt_realPrec(r: Real, prec: Int): Text
   Convert a real to its printable form.  Use a maximum precision of
   "prec"
```

The `readPrec` routine was added to allow real numbers to be formatted with a fixed precision.

E.3.3 lex

```
lex_failure: Exception
lex_scan(r: Rd, t: Text): Text ! rd_failure thread_alerted
   Read from r the longest prefix formed of characters listed in t, and
   return it.
lex_scanNonBlanks(r: Rd): Text ! rd_failure thread_alerted
   Read from r the longest prefix formed of characters nonblank
   characters, which means any in the range {'!' .. '~'), and return
```

```
  it.
lex_skip(r: Rd, t: Text): Ok ! rd_failure thread_alerted
  Read from r the longest prefix formed of characters listed in t, and
  discard it.
lex_skipBlanks(r: Rd): Ok ! rd_failure thread_alerted
  Read from r the longest prefix formed of blanks, which means any of
  {' ', '', '

lex_match(r: Rd, t: Text): Ok ! lex_failure rd_failure thread_alerted
  Read from r the string t and discard it; raise failure if not found.
lex_bool(r: Rd): Bool ! lex_failure rd_failure thread_alerted
  Skip blanks, and attempt to read a boolean from r.
lex_int(r: Rd): Int ! lex_failure rd_failure thread_alerted
  Skip blanks, and attempt to read an integer from r.
lex_real(r: Rd): Real ! lex_failure rd_failure thread_alerted
  Skip blanks, and attempt to read a real from r.
```

The `scan`, `scanNonBlanks`, `skip`, and `skipBlanks` routines were added to make this module more useful.

E.3.4 net

```
net_failure: Exception
All(T)  net_who(o: T): Text ! net_failure thread_alerted
  Return a text indicating where a network object or engine is
  registered, or the empty text if the argument is an object that has
  not been registered with a name server.
All(T<:{}) net_export(name: Text, server: Text, o: T): T
  ! net_failure thread_alerted
  Export an object under name 'name', to the name server at IP address
  'server'. The empty text denotes the local IP address.
Some(T<:{}) net_import(name: Text, server: Text): T
  ! net_failure thread_alerted
  Import the object of name 'name', from the name server at IP address
  'server'. The empty text denotes the local IP address.
All(T) net_exportEngine(name: Text, server: Text, arg: T): Ok
  ! net_failure thread_alerted
  Export an engine under name 'name', to the name server at IP address
  'server'. The empty text denotes the local IP address. The 'arg' is
  given as an argument to all procedures received by the engine to
  execute.
Some(T)All(U) net_importEngine(name: Text, server: Text): ((T)->U)->U
  ! net_failure thread_alerted
  Import the object of name 'name', from the name server at IP
  address 'server'. The empty text denotes the local IP address.
net_setSiteName(name: Text): Text ! net_failure thread_alerted
net_setDefaultSequencer(host name: Text): Ok ! net_failure
            thread_alerted
```

We added two routines to the `net` module to support the Shared Object runtime. The routine `setSiteName` is used to assign a symbolic name to the current process. The routine

`setDefaultSequencer` defines the sequencer for this process by specifying its symbolic name and the host on which it resides.

E.3.5 os

```
os_error: Exception
  A generic operating system exception, raise by various libraries.
os_type: Text
  A string describing the general type of this OS. Currently, one of
  "POSIX" or "WIN32".
os_target: Text
  The Modula-3 build target for this process.  Examples include
  "HPPA", "NT386", "SOLgnu", "SOLsun", "IRIX5, and "LINUXELF".
os_newPipe(): [Wr,Rd] ! os_error
  Create a new channel allowing bytes written to the "Wr" to be read
  from "Rd".
```

We added two constants to the `os` module, `type` and `target`. These are Modula-3 constants that identify the operating system as Unix (“POSIX”) or Windows (“WIN32”), and identify the specific variation of the operating system (for example, “HPPA” is HP-UX on the HPPA processor, and “LINUXELF” is Linux with ELF object files.) These variations are guidelines, as (for example) “NT386” is the only version of the Windows compiler for the x86 architecture, but it runs on both NT and Windows95.

We also added a function to create a local pipe.

E.3.6 process

```
process_new(pr: Processor, nameAndArgs: [Text], mergeOut: Bool,
          wd: Text): Process ! os_error
  Create a process from a processor and the given process name and
  arguments. The local processor is available as the lexically scoped
  identifier "processor". If mergeOut is true, use a single pipe for
  stdout and stderr. If "wd" is not "", it specifies the working
  directory for the process.
process_id(p: Process): Int
  Get the process id of the process 'p'
process_myId: Int
  The process id of this process
process_in(p: Process): Wr
  The stdin pipe of a process.
process_out(p: Process): Rd
  The stdout pipe of a process.
process_err(p: Process): Rd
```

```
  The stderr pipe of a process.
process_complete(p: Process): Int
  Wait for the process to exit, close all its pipes, and return the
  exit code
process_filter(pr: Processor, nameAndArgs: [Text], wd: Text,
               input:Text):Text ! net_failure os_error
  Create a process from a processor and the given process name and
  arguments. The local processor is available as the lexically scoped
  identifier "processor". The stderr output is merged to stdout.
  If "wd" is not "", it specifies the working directory for the
  process.
  Usage: feed the input to its stdin pipe and close it; read all the
  output from its stdout pipe and close it; return the output.
process_getWorkingDirectory(pr: Processor): Text ! os_error
  The current working directory of this process.
process_setWorkingDirectory(pr: Processor, dir: Text) ! os_error
  Change then current working directory of this process.
```

We added the facilities to get the ID of a process created with this interface, or of the Repo process specified by the `processor` argument. We also added the ability to get and set the current working directory of the Repo process specified by the `processor` argument.

E.3.7 sys

```
All(T) sys_copy(x: T): T ! net_failure
  (also 'copy(x)') Make a local copy of a value, including most
  distributed values.
sys_address: Text
  Return network the address of this process.
sys_getEnvVar(t: Text): Text
  Return the value of the env variable whose name is t, or "" if there
  is no such variable.
sys_paramCount: Int
  The number of program parameters.
sys_getParam(n: Int): Text
  Return the n-th program parameter (indexed from 0).
sys_callFailure: Exception
  Can be raised by Modula-3 code during a sys_call.
Some(T)Some(U) sys_call(name: Text, args: [T]): U ! sys_callFailure
  Call a pre-registered Modula-3 procedure.
sys_timeNow: Real
  The current time
sys_timeGrain: Real
  The time clock granularity
sys_timeShort(t: Real): Text
  A short formated representation of time "t"
sys_timeLong(t: Real): Text
  A long formated representation of time "t"
sys_registerExitor (proc: ()->ok): ok
  Provide a proc to be called when the process exits.  The proc takes
```

```
    no arguments and the return value is ignored.
```

We added the ability to retrieve environment variables and the system time (in both numeric and text formats), including an indication of the granularity of the time clock. We also added the ability to register Repo functions that should be executed when the process terminates.

E.3.8 text

```
t: Text
  A string in double quotes.
text_new(size: Int, init: Char): Text
  A text of size 'size', all filled with 'init'.
text_empty(t: Text): Bool
  Test for empty text.
text_length(t: Text): Int
  Length of a text.
text_equal(t1: Text, t2: Text): Bool
  Text equality (case sensitive).
text_char(t: Text, i: Int): Char
  The i-th character of a text (if it exists); zero-indexed.
text_sub(t: Text, start: Int, size: Int): Text
  The subtext beginning at 'start', and of size 'size' (if it exists).
text_&(t1: Text, t2: Text): Text
  (also infix '&') The concatenation of two texts.
text_precedes(t1: Text, t2: Text): Bool
  Whether 't1' precedes 't2' in lexicographic (ascii) order.
text_decode(t: Text): Text
  Every occurrence of an escape sequence is replaced by the
  corresponding non-printing formatting character: \\ = \;  \' = ';
  \" = "; \n = LF; \r = CR; \t = HT; \f = FF; \t = HT;
  \xxx = xxx (octals 000..177); \c = c (otherwise).
text_encode(t: Text): Text
  Every occurrence of a non-printing formatting character is replaced
  by an escape sequence.
text_explode(seps: Text, t: Text): [Text]
  Splits an input text into a similarly ordered array of texts, each a
  maximal subsequence of the input text not containing sep chars. The
  empty text is exploded as a singleton array of the empty text. Each
  sep char in the input produces a break, so the size of the result is
  1 + the number of sep chars in the text.
  implode(explode( "c",text),'c') is the identity.
text_implode(sep: Char, a: [Text]): Text ! net_failure
  Concatenate an array of texts into a single text, separating the
  pieces by a single sep char. A zero-length array is imploded as the
  empty text. explode("c",implode( 'c',text)) is the identity
  provided that the array has positive size and sep does not occur in
  the array elements.
text_hash(t: Text): Int
  A hash function.
text_toInt(t: Text): Int
```

```
  Convert a text to an integer (see also fmt_).
text_fromInt(n: Int): Text
  Convert an integer to a text (see also lex_).
text_findFirstChar(c: Char, t: Text, n: Int): Int
  The index of the first occurrence of 'c' in 't', past 'n'. -1 if not
  found.
text_findLastChar(c: Char, t: Text, n: Int): Int
  The index of the last occurrence of 'c' in 't', before 'n'. -1 if
  not found.
text_findFirst(p: Text, t: Text, n: Int): Int
  The index of the first char of the first occurrence of 'p' in 't',
  past 'n'. -1 if not found.
text_findLast(p: Text, t: Text, n: Int): Int
  The index of the first char of the last occurrence of 'p' in 't',
  before 'n'. -1 if not found.
text_replaceAll(old: Text, new: Text, t: Text): Text
  Replace all occurrences of 'old' by 'new' in 't', as found by
  iterating 'findFirst'.
text_toUpper(t: Text): Int
  Return a text with all the lower case letters converted to upper
  case ones.
text_toLower(t: Text): Int
  Return a text with all the upper case letters converted to lower
  case ones.
```

We added the conversions to upper or lower case.

E.3.9 thread

```
thread_mutex(): Mutex
  (also 'mutex()') A new mutex.
thread_condition(): Condition
  (also 'condition()') A new condition.
Some(T) thread_self(): Thread(T)
  The current thread.
thread_id(th: Thread(T)): Int
  The id of the thread.
thread_yield(): Ok
  If there are other threads ready to run, transfer control to one
  of them; otherwise continue with the current thread.
  Implementation note: the exact semantics of "yield" varies widely
  from system to system.  You shouldn't use it without consulting the
  detailed documentation for your implementation.
All(T) thread_fork(f: ()->T, stackSize: Int): Thread(T)
  (also 'fork(f,n)') Fork a new thread executing f. If stackSize is
  zero, a small default size is used.
All(T) thread_join(th: Thread(T)): T
  (also 'join(th)') Wait for a thread to complete, and return the
  result of its procedure.
thread_wait(mx: Mutex, cd: Condition): Ok
  (also 'wait(mx,cd)') Wait on a mutex and a condition.
thread_acquire(mx: Mutex): Ok
  Acquire a mutex (use lock ... end instead).
thread_release(mx: Mutex): Ok
```

```
    Release a mutex (use lock ... end instead)
  thread_broadcast(cd: Condition): Ok
    (also 'broadcast(cd)') Wake-up to all threads waiting on a
    condition.
  thread_signal(cd: Condition): Ok
    (also 'signal(cd)') Wake-up at least one thread waiting on a
    condition.
  thread_pause(r: Real): Ok
    (also 'pause(r)') Pause the current thread for r seconds.
  All(T) thread_lock(m: Mutex, body: ()->T): T
    Execute under a locked mutex (use lock ... end instead).
  thread_alerted: Exception
    (See the threads spec.)
  All(T) thread_alert(t: Thread(T)): Ok
    (See the threads spec.)
  thread_testAlert(): Bool
    (See the threads spec.)
  thread_alertWait(mx: Mutex, cd: Condition): Ok ! thread_alerted
    (See the threads spec.)
  All(T) thread_alertJoin(th: Thread(T)): Ok ! thread_alerted
    (See the threads spec.)
  thread_alertPause(r: Real): Ok ! thread_alerted
    (See the threads spec.)
  thread_pool(maxThreads maxIdleThreads stackSize: int): WorkerPool
    create a new thread worker pool, with at most maxThreads active
    threads, maxIdleThreads idle threads.  If stackSize is zero, a
    small default size is used.
  thread_addWork(pool: WorkerPool, work: ()->ok): Ok
    add a piece of work to the work queue for the thread pool.  work
    is represented by a procedure that performs the work
  thread_stealWorker(pool: WorkerPool): Bool
    steal a worker thread from a worker pool.  Removes the current
    thread from the list of threads performing work for the pool
    (allowing another to be created).   If a piece of work will require
    a thread to be idle for a long period of time,  this function can be
    called.
  thread_finish(pool: WorkerPool): Ok
    wait for all the work in the thread pool work queue to be finished.
```

We added support to the `thread` module for thread pools, as described in Section 3.4.2.1. A thread pool is created with `thread_pool`, and work objects are added to the pool's work queue with `thread_addWork`. Work is represented as a function closure with no arguments whose return value is ignored.

E.4 Unchanged Modules

The modules in this section were present in Obliq and have not been changed in Repo. They are included here for reference.

E.4.1 bool

```
true: Bool
  The constant true.
false: Bool
  The constant false.
All(T)All(U) bool_is(x: T, y: U): Bool
  (also infix 'is') Identity predicate: value equality for
  Ok, Bool, Int, Real, Char, Text, Exception; pointer equality
  otherwise.
All(T)All(U) bool_isnot(x: T, y: U): Bool
  (also infix 'isnot') Negation of 'is'.
bool_not(b: Bool): Bool
  (also 'not(b)')
bool_and(b1: Bool, b2: Bool): Bool
  (also infix 'and')
bool_or(b1: Bool, b2: Bool): Bool
  (also infix 'or')
```

E.4.2 char

```
c: Char
  A character in single quotes.
ascii_char(n: Int): Char
  The ascii character of integer code 'n'.
ascii_val(c: Char): Int
  The integer code of the ascii character 'c'.
```

E.4.3 color

```
color_named(name: Text): Color
  Get a color from its name (see the ColorName M3 interface).
color_rgb(r: Real, g: Real, b: Real): Color
  Get a color from rgb (each 0.0 .. 1.0).
color_hsv(hr: Real, sr: Real, v: Real): Color
  Get a color from hsv (each 0.0 .. 1.0).
color_r(c: Color): Real
  The red color component.
color_g(c: Color): Real
  The green color component.
color_b(c: Color): Real
  The blue color component.
color_h(c: Color): Real
  The hue color component.
color_s(c: Color): Real
  The saturation color component.
color_v(c: Color): Real
```

```
  The value color component.
color_brightness(c: Color): Real
  The total brightness (0.0 .. 1.0).
```

E.4.4 form

```
form_failure: Exception
form_new(t: Text): Form ! form_failure
  Read a form description from a text.
form_fromFile(file: Text): Form ! form_failure thread_alerted
  Read a form description from a file.
form_attach(fv: Form, name: Text, f: (Form)->Ok): Ok ! form_failure
  Attach a procedure to an event, under a form. The procedure is
  passed back the form when the event happens.
form_getBool(fv: Form, name: Text, property: Text): Bool !
         form_failure
  Get the boolean value of the property of the named interactor.
  (Do not confuse with form_getBoolean.)
form_putBool(fv: Form, name: Text, property: Text, b: Bool): Ok
  ! form_failure
  Set the boolean value of the named property of the named interactor.
  (Do not confuse with form_putBoolean.)
form_getInt(fv: Form, name: Text, property: Text): Int ! form_failure
  Get the integer value of the named property of the named interactor.
  If property is the empty text, get the ÒvalueÓ property.
form_putInt(fv: Form, name: Text, property: Text, n: Int): Ok
       ! form_failure
  Set the integer value of the named property of the named interactor.
  If property is the empty text, set the ÒvalueÓ property.
form_getText(fv: Form, name: Text, property: Text): Text !
       form_failure
  Get the text value of the named property of the named interactor. If
  property is the empty text, get the ÒvalueÓ property.
form_putText(fv: Form, name: Text, property: Text, t: Text,
            append: Bool): Ok ! form_failure
  Set the text value of the named property of the named interactor. If
  property is the empty text, set the ÒvalueÓ property.
form_getBoolean(fv: Form, name: Text): Bool ! form_failure
  Get the boolean value of the named boolean-choice interactor.
form_putBoolean(fv: Form, name: Text, b: Bool): Ok ! form_failure
  Set the boolean value of the named boolean-choice interactor.
form_getChoice(fv: Form, radioName: Text): Text ! form_failure
  Get the choice value of the named radio interactor.
form_putChoice(fv: Form, radioName: Text, choiceName: Text): Ok
  ! form_failure
  Set the choice value of the named radio interactor.
form_getReactivity(fv: Form, name: Text): Text ! form_failure
  Get the reactivity of the named interactor. It can be "active",
  "passive", "dormant", or "vanished".
form_putReactivity(fv: Form, name: Text, r: Text): Ok ! form_failure
  Set the reactivity of the named interactor. It can be "active",
  "passive", "dormant", or "vanished".
form_popUp(fv: Form, name: Text): Ok ! form_failure
  Pop up the named interactor.
```

```
form_popDown(fv: Form, name: Text): Ok ! form_failure
  Pop down the named interactor.
form_numOfChildren(fv: Form, parent: Text): Int ! form_failure
  Return the number of children of parent.
form_child(fv: Form, parent: Text, n: Int): Text ! form_failure
  Return the n-th child of parent.
form_childIndex(fv: Form, parent: Text, child: Text): Int !
           form_failure
  Return the index of the given child of parent.
form_insert(fv: Form, parent: Text, t: Text, n: Int): Ok !
           form_failure
  Insert the form described by t as child n of parent.
form_move(fv: Form, parent: Text, child: Text, toChild: Text,
         before: Bool): Ok ! form_failure
  Move child before or after toChild of parent; after "" means first,
  before "" means last.
form_delete(fv: Form, parent: Text, child: Text): Ok ! form_failure
  Delete the named child of parent.
form_deleteRange(fv: Form, parent: Text, n: Int, count: Int): Ok
  ! form_failure
  Delete count children of parent, from child n.
form_takeFocus(fv: Form, name: Text, select: Bool): Ok ! form_failure
  Make the named interactor acquire the keyboard focus, and optionally
  select its entire text contents.
form_show(fv: Form): Ok ! form_failure
  Show a window containing the form on the default display.
form_showAt(fv: Form, at: Text, title: Text): Ok ! form_failure
  Show a window containing the form on a display. For an X display,
  at=<machine name>(':'|'::')<num>(''|'.'<num>); at="" is the default
  display. The title is shown in the window header.
form_hide(fv: Form): Ok ! form_failure
  Hide the window containing the form.
```

This module is used to manipulate the Modula-3 Trestle windowing system

[Manasse, 1993].

E.4.5 int

```
n: Int
  Positive integer constants.
~n: Int
  Negative integer constants.
int_minus(n: Int): Int
    Integer negation.
int_+(n1: Int, n2: Int): Int
  Integer addition.
int_-(n1: Int, n2: Int): Int
  Integer difference.
int_*(n1: Int, n2: Int): Int
  Integer multiplication.
int_/(n1: Int, n2: Int): Int
  Integer division.
int_%(n1: Int, n2: Int): Int
```

```
  (also infix '%') Integer modulo.
int_<(n1: Int, n2: Int): Bool
  Integer less-than predicate.
int_>(n1: Int, n2: Int): Bool
  Integer greater-than predicate.
int_<=(n1: Int, n2: Int): Bool
  Integer no-greater-than predicate.
int_>=(n1: Int, n2: Int): Bool
  Integer no-less-than predicate.
```

E.4.6 math

```
math_pi: Real
  3.14159265358979323846264338 33.
math_e: Real
  2.71828182845904523536028747 14.
math_degree: Real
  0.01745329251994329576923690768 4; 1 degree in radiants.
math_exp(n: Real): Real
  e to the n-th power.
math_log(n: Real): Real
  log base e.
math_sqrt(n: Real): Real
  Square root.
math_hypot(n: Real, m: Real): Real
  sqrt((n*n)+(m*m)).
math_pow(n: Real, m: Real): Real
  n to the m-th power.
math_cos(n: Real): Real
  Cosine in radians.
math_sin(n: Real): Real
  Sine in radians.
math_tan(n: Real): Real
  Tangent in radians.
math_acos(n: Real): Real
  Arc cosine in radians.
math_asin(n: Real): Real
  Arc sine in radians.
math_atan(n: Real): Real
  Arc tangent in radians.
math_atan2(n: Real, m: Real): Real
  Arc tangent of n/m in radians.
```

E.4.7 online

```
All(T) sys_print(x: T, depth: Int): Ok
  Print an arbitrary value to stdout, up to some print depth. (Only
  available on-line.)
sys_printText(t: Text): Ok
```

```
  Print a text to stdout. (Only available on-line.)
sys_printFlush(): Ok
  Flush stdout. (Only available on-line.)
sys_pushSilence(): Ok
  Push the silence stack; when non-empty nothing is printed. (Only
  available on-line.)
sys_popSilence(): Ok
  Pop the silence stack (no-op on empty stack). (Only available
  on-line.)
sys_setPrompt(first: Text, next: Text): Ok
  Set the interactive prompts (defaults: first="- ", next="  "). (Only
  available on-line.)
sys_getSearchPath(): Text
  Get the current search path for 'load' and such. (Only available
  on-line.)
sys_setSearchPath(t: Text): Ok
  Set the current search path for 'load' and such. (Only available
  on-line.)
```

E.4.8 pickle

```
pickle_failure: Exception
All(T) pickle_write(w: Wr, v: T): Ok
  ! pickle_failure wr_failure thread_alerted
  Copy a value to a writer, similarly to sys_copy.
Some(T) pickle_read(r: Rd): T
  ! pickle_failure rd_failure rd_eofFailure thread_alerted
  Copy a value from a reader, similarly to sys_copy.
```

E.4.9 rd

```
rd_failure: Exception
rd_eofFailure: Exception
rd_new(t: Text): Rd
  A reader on a text (a Modula-3 TextRd).
rd_stdin: Rd
  The standard input (the Modula-3 Stdio.Stdin).
rd_open(fs: FileSystem, t: Text): Rd ! rd_failure
  Given a file system and a file name, returns a reader on a file
  (a Modula-3 FileRd, open for read). The local file system is
  available through the predefined lexically scoped identifier
  "fileSys". Moreover, "fileSysReader" is a read-only local file
  system.
rd_getChar(r: Rd): Char ! rd_failure rd_eofFailure thread_alerted
  Get the next character from a reader.
rd_eof(r: Rd): Bool ! rd_failure thread_alerted
  Test for the end-of-stream on a reader.
rd_unGetChar(r: Rd): Ok
  Put the last character obtained by getChar back into the reader
```

```
  (unfortunately, it may crash if misused!).
rd_charsReady(r: Rd): Int ! rd_failure
  The number of characters that can be read without blocking.
rd_getText(r: Rd, n: Int): Text ! rd_failure thread_alerted
  Read the next n characters, or at most n on end-of-file.
rd_getLine(r: Rd): Text ! rd_failure rd_eofFailure thread_alerted
  Read the next line and return it without including the end-of-line
  character.
rd_index(r: Rd): Int
  The current reader position.
rd_length(r: Rd): Int ! rd_failure thread_alerted
  Length of a reader (including read part).
rd_seek(r: Rd, n: Int): Ok ! rd_failure thread_alerted
  Reposition a reader.
rd_close(r: Rd): Ok ! rd_failure thread_alerted
  Close a reader.
rd_intermittent(r: Rd): Bool
  Whether the reader is stream-like (not file-like).
rd_seekable(r: Rd): Bool
  Whether the reader can be repositioned.
rd_closed(r: Rd): Bool
  Whether the reader is closed.
```

E.4.10 real

```
n.m: Int
  Positive real constants; m is optional.
~n.m: Int
  Negative real constants; m is optional.
real_minus(n: Real): Real
  (also '-n') Real negation.
real_minus(n: Int): Int
  (also '-n') Overloaded integer negation.
real_+(n1: Real, n2: Real): Real
  (also infix '+') Real addition.
real_+(n1: Int, n2: Int): Int
  (also infix '+') Overloaded integer addition.
real_-(n1: Real, n2: Real): Real
  (also infix '-') Real difference.
real_-(n1: Int, n2: Int): Int
  (also infix '-') Overloaded integer difference.
real_*(n1: Real, n2: Real): Real
  (also infix '*') Real multiplication.
real_*(n1: Int, n2: Int): Int
  (also infix '*') Overloaded integer multiplication.
real_/(n1: Real, n2: Real): Real
  (also infix '/') Real division.
real_/(n1: Int, n2: Int): Int
  (also infix '/') Overloaded integer division.
real_<(n1: Real, n2: Real): Bool
  (also infix '<') Real less-than predicate
real_<(n1: Int, n2: Int): Bool
  (also infix '<') Overloaded integer less-than predicate
real_>(n1: Real, n2: Real): Bool
```

```
  (also infix '>') Real greater-than predicate
real_>(n1: Int, n2: Int): Bool
  (also infix '>') Overloaded integer greater-than predicate
real_<=(n1: Real, n2: Real): Bool
  (also infix '<=') Real no-greater-than predicate
real_<=(n1: Int, n2: Int): Bool
  (also infix '<=') Overloaded integer no-greater-than pred.
real_>=(n1: Real, n2: Real): Bool
  (also infix '>=') Real no-less-than predicate.
real_>=(n1: Int, n2: Int): Bool
  (also infix '>=') Overloaded integer no-less-than pred.
real_float(n: Int): Real
  (also 'float(n)') Integer-to-real conversion.
real_float(n: Real): Real
  (also 'float(n)') Overloaded; identity on reals.
real_round(n: Real): Int
  (also 'round(n)') Real-to-integer rounding.
real_round(n: Int): Int
  (also 'round(n)') Overloaded; identity on integers.
real_floor(n: Real): Int
  Greatest integers no greater than n.
real_floor(n: Int): Int
  Overloaded; identity on integers.
real_ceiling(n: Real): Int
  Least integers no less than n.
real_ceiling(n: Int): Int
  Overloaded; identity on integers.
real_isNaN(n: Real): Bool
  Overloaded; false on integers.
```

E.4.11 vbt

```
vbt_failure: Exception
vbt_mu: Mutex
vbt_show(vbt: VBT): Ok
```

This module is also part of the Trestle Window System interface, as described in Section E.4.4.

E.4.12 wr

```
wr_failure: Exception
wr_new(): Wr
  A writer to a text (a Modula-3 TextWr).
wr_toText(w: Wr): Text
  Emptying a writer to a text..
wr_stdout: Wr
  The standard output (the Modula-3 Stdio.Stdout).
```

```
wr_stderr: Wr
   The standard error (the Modula-3 Stdio.Stderr).
wr_open(fs: FileSystem, t: Text): Wr ! wr_failure
   Given a file system and a file name, returns a writer to the
   beginning of a file (a Modula-3 FileWr, open for write). The local
   file system is available through the predefined lexically scoped
   identifier "fileSys".
wr_openAppend(fs: FileSystem, t: Text): Wr ! wr_failure
   Given a file system and a file name, returns a writer to the end of
   file (a Modula-3 FileWr, open for append). The local file system is
   available through the predefined lexically scoped identifier
   "fileSys".
wr_putChar(w: Wr, c: Char): Ok ! wr_failure thread_alerted
   Put a character to a writer .
wr_putText(w: Wr, t: Text): Ok ! wr_failure thread_alerted
   Put a text to a writer .
wr_flush(w: Wr): Ok ! wr_failure thread_alerted
   Flush a writer: all buffered writes to their final destination.
wr_index(w: Wr): Int
   The current writer position
wr_length(w: Wr): Int ! wr_failure thread_alerted
   Length of a writer.
wr_seek(w: Wr, n: Int): Ok ! wr_failure thread_alerted
   Reposition a writer.
wr_close(w: Wr): Ok ! wr_failure thread_alerted
   Close a writer.
wr_buffered(w: Wr): Bool
   Whether the writer is buffered.
wr_seekable(w: Wr): Bool
   Whether the writer can be repositioned.
wr_closed(w: Wr): Bool
   Whether the writer is closed.
```

APPENDIX F Another Replicated Mutex

In Section 4.6.4 we discussed the design of a distributed mutex, and presented a simple implementation in Figure 4-8. That version suffered from the problem that the mutex would not be released if the process containing the holder ended without unlocking the mutex.

In this appendix, we present a slightly more complex version, to show one way this problem can be overcome. In this version, the `id` method now returns a client-server object with a field containing the textual id of the client. When the lock is acquired, all clients request notification if the id object becomes unreachable, using the `unreachable` statement (in the `enqueueId` method). Objects typically become unreachable when the process that contains them terminates.

The `unreachable` statement takes a procedure argument that will be executed when the system determines that the object is unreachable. This procedure is executed in all copies of the mutex, and releases the lock held by the now unreachable client. It is safe to execute this in all copies because the first release of the mutex will succeed, and subsequent releases will quietly fail. Since all clients are watching for disconnection, the likelihood of one of them noticing in a timely manner increases.

Since the runtime only checks sites for disconnection when a method call is made to that site, or every few minutes if no calls are made, we added a facility to speed up the process. If a site executes the `startWatcher` method, a thread will be forked that polls the current mutex holder every second. Therefore, if that process dies, the thread will be guaranteed to notice is less than a second, and release the lock.

F.1 mutex.obl

```
module mutex;

(* exceptions raised when release and acquire are used incorrectly *)
let unheld = exception("unheld mutex");
let held = exception("held mutex");

let new = proc ()
     let ret = {replicated,
       (* create a return a client-server object that represents this
         process *)
       id => meth (s)
            {txt => sys_address & "." & fmt_int(process_myId) & "."&
                   fmt_int(thread_id(thread_self()))
            };
         end,

       (* variables used to control the mutex. A condition variable,
         and the current holder object and their text id *)
       cv => thread_condition(),
       holder => ok,
       holderId => ok,

       (* a utility routine to start a watcher thread that polls the
         current holder, to see if they are still alive *)
       startWatcher => meth (s)
            thread_fork (proc ()
                 try
                   loop
                     thread_alertPause(1.0);
                     if s.holder isnot ok then
                       try
                         s.holder.txt;
                       except net_failure =>
                         s.dequeueId(s.holderId);
                       end;
                     end;
                   end;
                 except thread_alerted => end;
                 ok;
               end, 50000);
         end,

       (* internal update methods, that enqueues/dequeues the current id
         in the mutex. The names are a holdover from an earlier
         implementation. enqueueId succeeds if the mutex is free,
         fails otherwise. dequeueId succeeds if the current thread
         holds the mutex, fails otherwise. *)
       enqueueId => umeth (s,id,txt)
            if s.holder isnot ok then raise (held) end;
            s.holder := id;
            s.holderId := txt;
            unreachable id do
              proc (o,st)
                   try
                     s.dequeueId(txt);
                   except unheld => end;
```

```
                end;
          end,
        dequeueId => umeth (s,txt)
             if s.holderId isnot txt then raise(unheld) end;
             s.holder := ok;
             s.holderId := ok;
             thread_signal(s.cv);
          end

        (* acquire the mutex, block until successful *)
        acquire => meth (s)
             let id = s.id;
             if s.holder isnot ok then
               if s.holderId is id.txt then raise(held) end;
             end;
             watch s.cv until
               try
                 s.enqueueId(id, id.txt);
               except held => end;
               s.holderId is id.txt;
             end;
          end,
        (* try to acquire the mutex, return if successful, raise an
          exception if not successful (already locked) *)
        tryAcquire => meth (s)
             let id = s.id;
             s.enqueueId(id, id.txt);
          end,
        (* release the mutex, return is successful, raise an exception
          if not held by this thread *)
        release => meth (s)
             s.dequeueId(s.holderId);
          end,
      };

      (* define the pickler for the mutex object. Send holder and holderId
        across the network, but recreate a new condition variable at the
        new site. *)
      objectpickler ret
          reader {simple,
                  cv => meth (s,c) thread_condition() end,
                  holder => meth (s,q) q end,
                  holderId => meth (s,q) q end}
          writer {simple,
                  cv => meth (s,c) ok end,
                  holder => meth (s,q) q end,
                  holderId => meth (s,q) q end};

      ret;
    end;

end module;
```

APPENDIX G Additional Enhancements To Repo-3D

Aside from being distributed, Repo-3D improves upon Obliq-3D in a number of ways. One important enhancement is in the area of performance; DistAnim-3D is significantly faster than Anim-3D, for two reasons. First, DistAnim-3D caches the internal representation of scene subgraphs (i.e., using OpenGL display lists or Renderware Clumps), including keeping multiple caches when a subgraph is attached to one or more scenes in multiple locations. These caches are only rebuilt as necessary. Anim-3D, on the other hand, does no caching, rerendering the scene in immediate mode every frame. Second, DistAnim-3D keeps track of when and where scene damage occurs (or may occur, in the case of synchronous and asynchronous properties) and only examines those parts of the graph to see if the caches need to be rebuilt. Anim-3D, on the other hand, examines the entire graph, and all properties, before each frame.

In addition to improving the performance, Repo-3D also increases the functionality of Obliq-3D by the addition of new GOs and properties. The new GOs address specific needs of our domain:

- Choice groups are group nodes that display only one of their children, rather than all of them. They are needed to allow efficient alternate representations of a subgraph, since it is possible to implement them so that changing the choice of which subgraph to use is significantly more efficient than replacing the subtree of a normal group node, primarily because the internal caches for all the subnodes can be prebuilt.
- Text objects allow text to be rendered more efficiently than if it was built using other Repo-3D facilities, and allow the use of properties to specify the details of the text objects. 2D text objects render text as flat bitmaps in the plane of the screen, allowing us to present readable text labels to the user.

- Indexed line and polygon sets allow complex models to be used. Creating complex scenes one polygon or line at a time (using Obliq-3D's polygon and line objects) results in extremely inefficient scenes (both in terms of time and space), and is also limiting, as multiple adjoining polygon objects are not smoothly shaded across their boundaries.

When designing these additional GOs, we attempted to maintain the high level of flexibility that made Obliq-3D unique. For example, indexed line and polygon sets can use dynamic point properties for all of their 3D points, making it easy to create polygonal objects that deform over time, or in reaction to their environment.

In addition to these new GOs, we added a number of properties to the system, most notably support for texture maps on all GOs that inherit from the `SurfaceGO` object. Repo-3D also introduces a new class of callback objects that may be attached to nodes by a programmer, to monitor the location of the origin of the node's coordinate system; *transform* callbacks are used to monitor the transformed 3D position of the node, while *projection* callbacks are used to monitor the 2D projection of that node on the root window. For the details of the Repo-3D location callback modules, see Appendix H.5. Each time the scene is refreshed, the object's callback method is invoked with the current 3D or 2D information.

These callbacks were created because we often found ourselves needing these two pieces of information in the programs we were building, but the declarative nature of the graphics library made this information hard to obtain in a straightforward manner. For example, to integrate 2D windows into the 3D worlds presented by Coterie, as seen in the Nynex crossbox maintenance prototype of Section 2.6 (shown in Figure 2-8), an empty group node is placed at the 3D position of the 2D window, and a projection callback is attached to that group. Each time the projected position changes, the 2D window is moved to the new position. The projection callbacks are also used in the Touring Machine prototype, both to determine the color of the labels and to construct the arrow at the bottom of the screen that points at the currently selected building.

One final change between Obliq-3D and Repo-3D concerns synchronous properties. In Obliq-3D, when an animation handle is signaled and completes the animations for

all attached synchronous properties, it flushes the requests from the synchronous behaviors of those properties. Therefore, to repeat an animation, a programmer must then reinitialize those behaviors with the same set of requests. Furthermore, it was impossible to flush requests from a synchronous behavior without signalling the animation and waiting for it to complete. Our programmers found both of these design choices to be a nuisance, so we added a command to each of the synchronous properties to flush their current requests, and modified the animation handle so it did not flush requests from the synchronous behaviors.

APPENDIX H Repo-3D Modules

In Chapter 5, we described different components of Repo-3D, and included excerpts from the help files of a few of the Repo-3D modules. In this appendix, we include the help files for all the Repo-3D modules, and provide occasional clarification when the purpose of a module is not clear from the name, both as a reference and to give the curious reader an idea of the scope of the library.

H.1 Graphics Objects

The graphical objects in Repo-3D all inherit from the abstract GO object, and use the GOCB object for change notification.

H.1.1 GO

```
   GO_PropUndefined: Exception
   GO_StackError: Exception
   GO_ListError: Exception
   GO_PropError: Exception
   GO_Transform: TransformPropName
   GO_SetTransform(go: GO, xf: TransformVal): Ok ! replica_failure
                  PropError
   GO_GetTransform(go: GO): TransformPropVal ! GO_PropUndefined
                  replica_failure
   GO_Pickable: TransformPropName
   GO_SetPickable(go: GO, xf: BooleanPropVal): Ok ! replica_failure
                  PropError
   GO_GetPickable(go: GO): BooleanPropVal ! GO_PropUndefined
                  replica_failure
WHERE
   GO <: ProxiedObj &
      { setProp: (PropName,PropVal) => Ok ! GO_PropError replica_failure,
        unsetProp: (PropName) => Ok ! GO_PropUndefined replica_failure,
        getProp: (PropName) => PropVal ! GO_PropUndefined
                                  replica_failure,
        setName: (Text) => Ok ! replica_failure,
        getName: () => Text ! replica_failure,
```

```
        findName: (Text) => GO ! replica_failure,
        setLocalProp: (PropName,PropVal) => Ok ! GO_PropError
                                  replica_failure,
        unsetLocalProp: (PropName) => Ok ! GO_PropUndefined GO_PropError
                                  replica_failure,
        getLocalProp: (PropName) => PropVal ! GO_PropUndefined
                                  replica_failure,
        hideGlobalProp: (PropName) => Ok ! GO_PropError replica_failure,
        revealGlobalProp: (PropName) => Ok ! GO_PropUndefined
                                  GO_PropError, replica_failure,
        isPropHidden: (PropName) => Bool ! replica_failure,
        setLocalPropsGlobally: () => Ok ! GO_PropError replica_failure,
        pushMouseCB: (cb: MouseCB) => Ok ! replica_failure,
        popMouseCB: () => Ok ! GO_StackError ! replica_failure,
        removeMouseCB: (cb: MouseCB) => Ok ! GO_StackError
                                  replica_failure,
        invokeMouseCB: (mr: MouseRec) => Ok ! replica_failure,
        pushPositionCB: (cb: PositionCB) => Ok ! replica_failure,
        popPositionCB: () => Ok ! GO_StackError replica_failure,
        removePositionCB: (cb: PositionCB) => Ok ! GO_StackError
                                  replica_failure,
        invokePositionCB: (mr: PositionRec) => Ok ! replica_failure,
        pushKeyCB: (cb: KeyCB) => Ok ! replica_failure,
        popKeyCB: () => Ok ! GO_StackError replica_failure,
        removeKeyCB: (cb: KeyCB) => Ok ! GO_StackError replica_failure,
        invokeKeyCB: (mr: KeyRec) => Ok ! replica_failure,
        addProjectionCB: (cb: ProjectionCB) => Ok ! replica_failure,
        removeProjectionCB: (cb: ProjectionCB) => Ok ! GO_ListError
                                  replica_failure
        invokeProjectionCB: (mr: ProjectionRec) => Ok ! replica_failure,
        addTransformCB: (cb: TransformCB) => Ok ! replica_failure,
        removeTransformCB: (cb: TransformCB) => Ok ! GO_ListError
                                  replica_failure,
        invokeTransformCB: (mr: TransformRec) => Ok ! replica_failure,
        getBoundingVolumeCenter: () => Point3 ! replica_failure,
        getBoundingVolumeRadius: () => Real ! replica_failure }
  TransformVal = TransformPropVal + Matrix4
  BooleanVal = BooleanPropVal + Bool
```

H.1.2 GOCB

```
  GOCB_New(obj: GO, overrides: Obj): T;
  GOCB_Cancel(cbobj: T): T;
WHERE
  T <: {simple} & overrides;
  overrides contains one or more of these callback methods:
       pre`propagateLocalProps(obj: GO, add remove: [Prop_T]): Bool
       post`propagateLocalProps(obj: GO, add remove: [Prop_T]): Bool
       pre`setProp(obj: GO, prop: Prop_T): Bool
       post`setProp(obj: GO, prop: Prop_T): Bool
       pre`unsetProp(obj: GO, name: Prop_Name): Bool
       post`unsetProp(obj: GO, name: Prop_Name): Bool
       pre`setName(obj: GO, name: Text): Bool
       post`setName(obj: GO, name: Text): Bool
```

```
     pre`anyChange(obj: GO);
     post`anyChange(obj: GO);

If T is one of BoxGO, ConeGO, CylinderGO, DiskGO, SphereGO, TorusGO,
OrthoCameraGO, PerspCameraGO, AmbientLightGO, SpotLightGO,
PointLightGO, VectorLightGO, LineGO, Text2DGO, MarkerGO overrides
may also contain:
     pre`init(obj: GO): Bool
     post`init(obj: GO): Bool

If T is PolygonGO overrides may also contain:
     pre`init(obj: GO, pts: PointArray, s: GO_Shape): Bool
     post`init(obj: GO, pts: PointArray, s: GO_Shape): Bool

If T is QuadMeshGO overrides may also contain:
     pre`init(obj: GO, pts: [Point3], s: GO_Shape): Bool
     post`init(obj: GO, pts: [Point3], s: GO_Shape): Bool
     pre`addFacetColors(obj: GO, cols: [[Color]]): Bool
     post`addFacetColors(obj: GO, cols: [[Color]]): Bool
     pre`setColorOfFacet(obj: GO, i j: Int, col: Color): Bool
     post`setColorOfFacet(obj: GO, i j: Int, col: Color): Bool

If T is IndexedLineSetGO overrides may also contain:
     pre`init(obj: GO, pts: PointArray, index: [[Int]]): Bool
     post`init(obj: GO, pts: PointArray, index: [[Int]]): Bool
     pre`setColor(obj: GO, clr: ColorArray, cpv: Bool): Bool
     post`setColor(obj: GO, clr: ColorArray, cpv: Bool): Bool
     pre`setVertices (obj: GO, pts: [Point3]): Bool
     post`setVertices (obj: GO, pts: [Point3]): Bool

If T is IndexedPolygonSetGO overrides may also contain:
     pre`init(obj: GO, pts: PointArray, index: [[Int]],
                 creaseAngle: Real, s: GO_Shape): Bool
     post`init(obj: GO, pts: PointArray, index: [[Int]],
                 creaseAngle: Real, s: GO_Shape): Bool
     pre`setColor(obj: GO, clr: ColorArray, cpv: Bool): Bool
     post`setColor(obj: GO, clr: ColorArray, cpv: Bool): Bool
     pre`setVertices (obj: GO, pts: [Point3]): Bool
     post`setVertices (obj: GO, pts: [Point3]): Bool
     pre`setTexCoords(obj: GO, pts: Point2Array, idx: [[Int]]): Bool
     post`setTexCoords(obj: GO, pts: Point2Array, idx: [[Int]]): Bool

If T was created with provided normals, it may contain
     pre`initWithNormals(obj: GO, pts: [Point3], index: [[Int]],
                         normalPerVertex : Bool, normals : [Point3],
                         creaseAngle: Real, s: GO_Shape): Bool
     post`initWithNormals(obj: GO, pts: [Point3], index: [[Int]],
                         normalPerVertex : Bool, normals : [Point3],
                         creaseAngle: Real, s: GO_Shape): Bool

If T is GroupGO or ChoiceGroupGO overrides may also contain:
     pre`add(obj new: GO): Bool
     post`add(obj new: GO): Bool
     pre`remove(obj old: GO): Bool
     post`remove(obj old: GO): Bool
     pre`replace(obj old new: GO): Bool
     post`replace(obj old new: GO): Bool
     pre`flush(obj: GO): Bool
     post`flush(obj: GO): Bool
```

```
        pre`propagateLocalChildren(obj: GO, add remove: [GO]): Bool
        post`propagateLocalChildren(obj: GO, add remove: [GO]): Bool

   PointArray = [Point3] + [PointProp]
   Point2Array = [Point2] + [Point2Prop]
   ColorArray = [Color] + [ColorProp]
```

H.1.3 AmbientLightGO

```
   AmbientLightGO_New(c: ColorVal; int: RealVal): AmbientLightGO !
   GO_PropError
WHERE
   AmbientLightGO <: LightGO
   ColorVal = ColorPropVal + Color + Text
   RealVal = RealPropVal + Real + Int
```

For adding background ambient light to a scene.

H.1.4 BoxGO

```
   BoxGO_New(p1 p2: PointVal): BoxGO ! GO_PropError
   BoxGO_Corner1: PointPropName
   BoxGO_Corner2: PointPropName
   BoxGO_SetCorner1(o: GO, p: PointVal): Ok ! replica_failure
                                GO_PropError
   BoxGO_SetCorner2(o: GO, p: PointVal): Ok ! replica_failure
                                GO_PropError
WHERE
   BoxGO <: SurfaceGO
   PointVal = PointPropVal + Point3
```

H.1.5 CameraGO

```
   CameraGO_From: PointPropName
   CameraGO_To: PointPropName
   CameraGO_Up: PointPropName
   CameraGO_Aspect: PointPropName
   CameraGO_SetFrom(go: GO, PointVal): Ok ! replica_failure GO_PropError
   CameraGO_SetTo(go: GO, PointVal): Ok ! replica_failure GO_PropError
   CameraGO_SetUp(go: GO, PointVal): Ok ! replica_failure GO_PropError
   CameraGO_SetAspect(go: GO, RealVal): Ok ! replica_failure
                                GO_PropError
   CameraGO_Near: RealPropName;
   CameraGO_SetNear(go: GO, RealVal): Ok ! replica_failure GO_PropError
```

```
   CameraGO_Far: RealPropName;
   CameraGO_SetFar(go: GO, RealVal): Ok ! replica_failure GO_PropError
   CameraGO_FixedNear: BooleanPropName;
   CameraGO_SetFixedNear(go: GO, BoolVal): Ok ! replica_failure
                                GO_PropError
   CameraGO_FixedFar: BooleanPropName;
   CameraGO_SetFixedFar(go: GO, BoolVal): Ok ! replica_failure
                                GO_PropError
   CameraGO_Stereo: BooleanPropName;
   CameraGO_SetStereo(go: GO, BoolVal): Ok ! replica_failure
                                GO_PropError
   CameraGO_EyeSeparation: RealPropName;
   CameraGO_SetEyeSeparation(go: GO, RealVal): Ok ! replica_failure
                                GO_PropError
   CameraGO_FocalDistance: RealPropName;
   CameraGO_SetFocalDistance(go: GO, RealVal): Ok ! replica_failure
                                GO_PropError
TYPE
   CameraGO <: GO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   BoolVal = BooleanPropVal + Bool
```

The abstract base class for the various kinds of cameras. A camera is used to map a 3D world to a 2D window (represented by a RootGO).

H.1.6 ChoiceGroupGO

```
   ChoiceGroupGO_New(display: IntVal): ChoiceGroupGO
   ChoiceGroupGO_NewWithSizeHint(display: IntVal, size: Int):
                                ChoiceGroupGO
   ChoiceGroupGO_Display: IntPropName
   ChoiceGroupGO_SetDisplay(go: GO, child: IntVal): Ok
WHERE
   ChoiceGroupGO <: GroupGO
   IntVal = IntPropVal + Int
```

H.1.7 ConeGO

```
   ConeGO_New(base tip: PointVal, rad: RealVal): ConeGO
   ConeGO_NewWithPrec(base tip: PointVal, rad: RealVal, prec: Int):
                                ConeGO
   ConeGO_NewWithDoublePrec(base tip: PointVal, rad: RealVal,
                            prec1 prec2: Int): ConeGO
   ConeGO_RotationPrecision: IntPropName
   ConeGO_LengthPrecision: IntPropName
   ConeGO_Base: PointPropName
   ConeGO_Tip: PointPropName
```

```
   ConeGO_Radius: RealPropName
   ConeGO_SetRotationPrecision(o: GO, p: IntVal): Ok
   ConeGO_SetLengthPrecision(o: GO, p: IntVal): Ok
   ConeGO_SetBase(o: GO, p: PointVal): Ok
   ConeGO_SetTip(o: GO, p: PointVal): Ok
   ConeGO_SetRadius(o: GO, r: RealVal): Ok
WHERE
   ConeGO <: SurfaceGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   IntVal = IntPropVal + Int
```

H.1.8 CylinderGO

```
   CylinderGO_New(p1 p2: PointVal, rad: RealVal): CylinderGO
   CylinderGO_NewWithPrec(p1 p2: PointVal, rad: RealVal, prec: Int):
                                 CylinderGO
   CylinderGO_NewWithDoublePrec(p1 p2: PointVal, rad: RealVal,
                                 prec1 prec2: Int): CylinderGO
   CylinderGO_RotationPrecision: IntPropName
   CylinderGO_LengthPrecision: IntPropName
   CylinderGO_Point1: PointPropName
   CylinderGO_Point2: PointPropName
   CylinderGO_Radius: RealPropName
   CylinderGO_SetRotationPrecision(o: GO, p: IntVal): Ok
   CylinderGO_SetLengthPrecision(o: GO, p: IntVal): Ok
   CylinderGO_SetPoint1(o: GO, p: PointVal): Ok
   CylinderGO_SetPoint2(o: GO, p: PointVal): Ok
   CylinderGO_SetRadius(o: GO, r: RealVal): Ok
WHERE
   CylinderGO <: SurfaceGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   IntVal = IntPropVal + Int
```

H.1.9 DiskGO

```
   DiskGO_New(center normal: PointVal, rad: RealVal): DiskGO
   DiskGO_NewWithPrec(center normal: PointVal, rad: RealVal, prec: Int):
                                    DiskGO
   DiskGO_Precision: IntPropName
   DiskGO_Center: PointPropName
   DiskGO_Normal: PointPropName
   DiskGO_Radius: RealPropName
   DiskGO_SetPrecision(go: GO, prec: IntVal): Ok
   DiskGO_SetCenter(o: GO, p: PointVal): Ok
   DiskGO_SetNormal(o: GO, p: PointVal3): Ok
   DiskGO_SetRadius(o: GO, r: RealVal): Ok
WHERE
```

```
  DiskGO <: SurfaceGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
  IntVal = IntPropVal + Int
```

H.1.10 GroupGO

```
  GroupGO_BadElement: Exception
  GroupGO_New(): GroupGO
  GroupGO_NewWithSizeHint(size: Int): GroupGO
WHERE
  GroupGO <: GO &
    { add: (GO) => Ok ! GroupGO_BadElement replica_failure,
      remove: (GO) => Ok ! GroupGO_BadElement replica_failure,
      replace: (GO,GO) => Ok ! GroupGO_BadElement replica_failure,
      flush: () => Ok ! replica_failure,
      content: () => [GO] ! replica_failure,
      addLocal: (GO) => Ok ! GroupGO_BadElement replica_failure,
      removeLocal: (GO) => Ok ! GroupGO_BadElement replica_failure,
      replaceLocal: (GO,GO) => Ok ! GroupGO_BadElement replica_failure,
      removeLocalAddition: (GO) => Ok ! GroupGO_BadElement
                                replica_failure,
      removeLocalRemoval: (GO) => Ok ! GroupGO_BadElement
                                replica_failure,
      removeLocalReplacement: (GO) => Ok ! GroupGO_BadElement
                                replica_failure,
      flushLocal: () => Ok ! replica_failure,
      mergeLocalToGlobal: () => Ok ! replica_failure,
      localContent: () => [GO] ! replica_failure }
```

H.1.11 IndexedLineSetGO

```
  IndexedLineSetGO_BadVertexIndex   : Exception;
  IndexedLineSetGO_BadSize          : Exception;
  IndexedLineSetGO_NotAllowed       : Exception;
  IndexedLineSetGO_New(pts          : [PointVal],
                       index        : [[Int]]): IndexedLineSetGO

WHERE
  PointVal = PointPropVal + Point3

  IndexedLineSetGO <: GO & {
       setStaticColor ([Color], Bool) => Ok ! IndexedLineSetGO_BadSize
                 replica_failure,
       setDynamicColor ([ColorProp], Bool) => Ok !
                 IndexedLineSetGO_BadSize replica_failure,
       setVertices ([Point3]) => Ok ! IndexedLineSetGO_BadSize
                 IndexedLineSetGO_NotAllowed replica_failure
```

```
  }
```

H.1.12 IndexedPolygonSetGO

```
  IndexedPolygonSetGO_BadVertexIndex   : Exception;
  IndexedPolygonSetGO_BadSize          : Exception;
  IndexedPolygonSetGO_NormalNotAllowed : Exception;
  IndexedPolygonSetGO_NotAllowed       : Exception;
  IndexedPolygonSetGO_New(pts          : [PointVal],
                          index         : [[Int]],
                          creaseAngle: Real): IndexedPolygonSetGO
  IndexedPolygonSetGO_NewWithNormal(
                          pts             : [Point3],
                          index           : [[Int]],
                          normalPerVertex : Bool,
                          normals         : [Point3],
                          creaseAngle     : Real): IndexedPolygonSetGO
  IndexedPolygonSetGO_NewWithShapeHint(
                          pts        : [PointVal],
                          index      : [[Int]],
                          creaseAngle: Real,
                          shp        : Shape): IndexedPolygonSetGO
  IndexedPolygonSetGO_NewWithNormalAndShapeHint(
                          pts             : [Point3],
                          index           : [[Int]],
                          normalPerVertex : Bool,
                          normals         : [Point3],
                          creaseAngle     : Real,
                          shp             : Shape): IndexedPolygonSetGO
WHERE
  PointVal = PointPropVal + Point3

  IndexedPolygonSetGO <: SurfaceGO & {
       setStaticColor ([Color], Bool) => Ok !
                 IndexedPolygonSetGO_BadSize replica_failure,
       setDynamicColor ([ColorProp], Bool) => Ok !
                 IndexedPolygonSetGO_BadSize replica_failure,
       setStaticTexCoords ([Point2], [[Int]]) => Ok !
                 IndexedPolygonSetGO_BadSize replica_failure,
       setDynamicTexCoords ([Point2Prop], [[Int]]) => Ok !
                 IndexedPolygonSetGO_BadSize replica_failure,
       setVertices ([Point3]) => Ok ! IndexedPolygonSetGO_BadSize,
                 IndexedPolygonSetGO_NotAllowed replica_failure
  }
  Shape = Text  (one of "Unknown", "Convex", "NonConvex", "Complex")
```

H.1.13 LightGO

```
  LightGO_Color: ColorPropName
```

```
   LightGO_Switch: BooleanPropName
   LightGO_Intensity: RealPropName
   LightGO_SetColor(go: GO, c: ColorVal): Ok ! replica_failure
                  GO_PropError
   LightGO_SetSwitch(go: GO, b: BooleanVal): Ok ! replica_failure
                  GO_PropError
   LightGO_SetIntensity(go: GO, i: RealVal): Ok ! replica_failure
                  GO_PropError
WHERE
   LightGO <: GO
   ColorVal = ColorPropVal + Color + Text
   BooleanVal = BooleanPropVal + Bool
   RealVal = RealPropVal + Real
```

H.1.14 LineGO

```
   LineGO_New(p1 p2: PointVal): LineGO ! GO_PropError
   LineGO_Color: ColorPropName
   LineGO_Width: RealPropName
   LineGO_Type: LineTypePropName
   LineGO_Point1: PointPropName
   LineGO_Point2: PointPropName
   LineGO_SetColor(o: GO, c: ColorVal): Ok ! replica_failure
                  GO_PropError
   LineGO_SetWidth(o: GO, r: RealVal): Ok ! replica_failure GO_PropError
   LineGO_SetType(o: GO, t: LineType): Ok ! replica_failure GO_PropError
   LineGO_SetPoint1(o: GO, p: PointVal): Ok ! replica_failure
                  GO_PropError
   LineGO_SetPoint2(o: GO, p: PointVal): Ok ! replica_failure
                  GO_PropError
WHERE
   LineGO <: GO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   ColorVal = ColorPropVal + Color + Text
   LineTypeVal = LineTypePropVal + LineType
```

H.1.15 MarkerGO

```
   MarkerGO_New(point: PointVal): MarkerGO ! GO_PropError
   MarkerGO_Center: PointPropName
   MarkerGO_Color: ColorPropName
   MarkerGO_Scale: RealPropName
   MarkerGO_Type: MarkerTypePropName
   MarkerGO_SetCenter(o: GO, p: PointVal): Ok ! replica_failure
                  GO_PropError
   MarkerGO_SetColor(o: GO, c: ColorVal): Ok ! replica_failure
                  GO_PropError
   MarkerGO_SetScale(o: GO, r: RealVal): Ok ! replica_failure
```

```
                GO_PropError
   MarkerGO_SetType(o: GO, t: MarkerTypeVal): Ok ! replica_failure
                GO_PropError
WHERE
   MarkerGO <: GO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   ColorVal = ColorPropVal + Color + Text
   MarkerTypeVal = MarkerTypePropVal + MarkerType
```

H.1.16 OrthoCameraGO

```
   OrthoCameraGO_New(from to up: PointVal, height: RealVal):
                   OrthoCameraGO ! GO_PropError
   OrthoCameraGO_Height: RealPropName
   OrthoCameraGO_SetHeight(go: GO, height: RealVal): Ok !
                   replica_failure GO_PropError
WHERE
   OrthoCameraGO <: CameraGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
```

A camera that provides an orthographic projection of the world onto a RootGO.

H.1.17 PerspCameraGO

```
   PerspCameraGO_New(from to up: PointVal, fovy: RealVal): PerspCameraGO
                ! GO_PropError
   PerspCameraGO_Fovy: RealPropName
   PerspCameraGO_SetFovy(go: GO, fovy: RealVal): Ok ! replica_failure
                GO_PropError
WHERE
   PerspCameraGO <: CameraGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
```

A camera that provides a perspective projection of the world onto a RootGO.

H.1.18 PointLightGO

```
   PointLightGO_New(c: ColorVal, orig: PointVal,
          att0 att1 intensity: RealVal): PointLightGO ! GO_PropError
   PointLightGO_Origin: PointPropName
   PointLightGO_SetOrigin(go: GO, orig: PointVal): Ok ! replica_failure
          GO_PropError
```

```
   PointLightGO_Attenuation0: RealPropName
   PointLightGO_SetAttenuation0(go: GO, att: RealVal): Ok !
          replica_failure GO_PropError
   PointLightGO_Attenuation1: RealPropName
   PointLightGO_SetAttenuation1(go: GO, att: RealVal): Ok !
          replica_failure GO_PropError
WHERE
   PointLightGO <: LightGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   ColorVal = ColorPropVal + Color + Text
```

H.1.19 PolygonGO

```
   PolygonGO_New(pts: [PointVal]): PolygonGO
   PolygonGO_NewWithShapeHint(pts: [PointVal], s: Shape): PolygonGO
WHERE
   PolygonGO <: SurfaceGO
   PointVal = PointPropVal + Point3
   Shape = Text  (one of "Unknown", "Convex", "NonConvex", "Complex")
```

H.1.20 QuadMeshGO

```
   QuadMeshGO_BadSize: Exception
   QuadMeshGO_ColorsUndefined: Exception
   QuadMeshGO_New(pts: [[Point3]]): QuadMeshGO
   QuadMeshGO_NewWithShapeHint(pts: [[Point3]], s: Shape): QuadMeshGO
WHERE
   QuadMeshGO <: SurfaceGO &
          { addFacetColors: ([[Col]]) => Ok ! QuadMeshGO_BadSize,
            setColorOfFacet: (i j: Int, c: Col) => Ok !
                 QuadMeshGO_ColorsUndefined }
   Shape = Text  (one of "Unknown", "Convex", "NonConvex", "Complex")
   Col = Color + Text
```

H.1.21 RootGO

```
   RootGO_New(cam: CameraGO, base: GraphicsBase): RootGO
   RootGO_NewStd(): RootGO | GraphicsBase_Failure
   RootGO_NewStdWithBase(base: GraphicsBase): RootGO
   RootGO_Background: ColorPropName
   RootGO_DepthcueSwitch: BooleanPropName
   RootGO_DepthcueColor: ColorPropName
   RootGO_DepthcueFrontPlane: RealPropName
```

```
   RootGO_DepthcueBackPlane: RealPropName
   RootGO_DepthcueFrontScale: RealPropName
   RootGO_DepthcueBackScale: RealPropName
   RootGO_SetBackground(go: GO, c: ColorVal): Ok ! GO_PropError
   RootGO_SetDepthcueSwitch(go: GO, b: BooleanVal): Ok ! GO_PropError
   RootGO_SetDepthcueColor(go: GO, c: ColorVal): Ok ! GO_PropError
   RootGO_SetDepthcueFrontPlane(go: GO, r: RealVal): Ok ! GO_PropError
   RootGO_SetDepthcueBackPlane(go: GO, r: RealVal): Ok ! GO_PropError
   RootGO_SetDepthcueFrontScale(go: GO, r: RealVal): Ok ! GO_PropError
   RootGO_SetDepthcueBackScale(go: GO, r: RealVal): Ok ! GO_PropError
WHERE
   RootGO <: GroupGO &
        { changeCamera: (CameraGO) => Ok,
          getCamera: () => CameraGO,
          picking (x, y: Int) => [PickingInfo],
          addCameraTransformCB: (cb: TransformCB) => Ok,
          removeCameraTransformCB: (cb: TransformCB) => Ok !
                           GO_ListError,
          windowPosSize: () => PosSizeRec,
          changeTitle: (Text) => Ok,
          awaitDelete: () => Ok,
          destroy: () => Ok }
   Point2 = [2*Int]
   BooleanVal = BooleanPropVal + Bool
   RealVal = RealPropVal + Real + Int
   ColorVal = ColorPropVal + Color + Text
   PickingInfo = { gos: [ObGO.T],
                   minz, maxz: Int]}
```

A RootGO is a descendent of GroupGO, and serves as the root of a Repo-3D scene graph, manifesting itself as a 2D window. A camera maps the scene graph rooted at this object to the 2D image in the window.

H.1.22 SphereGO

```
   SphereGO_New(p: PointVal, rad: RealVal): SphereGO
   SphereGO_NewWithPrec(p: PointVal, rad: RealVal, prec: Int): SphereGO
   SphereGO_Precision: IntPropName
   SphereGO_Center: PointPropName
   SphereGO_Radius: RealPropName
   SphereGO_SetPrecision(go: GO, prec: IntVal): Ok
   SphereGO_SetCenter(go: GO, center: PointVal): Ok
   SphereGO_SetRadius(go: GO, radius: RealVal): Ok
WHERE
   SphereGO <: SurfaceGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   IntVal = IntPropVal + Int
```

H.1.23 SpotLightGO

```
  SpotLightGO_New(c: ColorVal, orig dir: PointVal,
                  conc spread att0 att1 int: RealVal):
         SpotLightGO ! GO_PropError
  SpotLightGO_Origin: PointPropName
  SpotLightGO_SetOrigin(go: GO, orig: PointVal): Ok ! replica_failure
         GO_PropError
  SpotLightGO_Direction: PointPropName
  SpotLightGO_SetDirection(go: GO, dir: PointVal): Ok ! replica_failure
         GO_PropError
  SpotLightGO_Concentration: RealPropName
  SpotLightGO_SetConcentration(go: GO, conc: RealVal): Ok !
         replica_failure GO_PropError
  SpotLightGO_SpreadAngle: RealPropName
  SpotLightGO_SetSpreadAngle(go: GO, spread: RealVal): Ok !
         replica_failure GO_PropError
  SpotLightGO_Attenuation0: RealPropName
  SpotLightGO_SetAttenuation0(go: GO, att: RealVal): Ok !
         replica_failure GO_PropError
  SpotLightGO_Attenuation1: RealPropName
  SpotLightGO_SetAttenuation1(go: GO, att: RealVal): Ok !
         replica_failure GO_PropError
WHERE
  SpotLightGO <: LightGO
  PointVal = PointPropVal + Point3
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
```

H.1.24 SurfaceGO

```
  SurfaceGO_Color: ColorPropName
  SurfaceGO_SetColor(o: GO, color: ColorVal): Ok ! replica_failure
         GO_PropError
  SurfaceGO_BackColor: ColorPropName
  SurfaceGO_SetBackColor(o: GO, color: ColorVal): Ok ! replica_failure
         GO_PropError
  SurfaceGO_RasterMode: RasterModePropName
  SurfaceGO_SetRasterMode(o: GO, t: RasterModeVal): Ok !
         replica_failure GO_PropError
  SurfaceGO_AmbientReflectionCoeff: RealPropName
  SurfaceGO_SetAmbientReflectionCoeff(o: GO, r: RealVal): Ok !
         replica_failure GO_PropError
  SurfaceGO_DiffuseReflectionCoeff: RealPropName
  SurfaceGO_SetDiffuseReflectionCoeff(o: GO, r: RealVal): Ok !
         replica_failure GO_PropError
  SurfaceGO_SpecularReflectionCoeff: RealPropName
  SurfaceGO_SetSpecularReflectionCoeff(o: GO, r: RealVal): Ok !
         replica_failure GO_PropError
  SurfaceGO_SpecularReflectionConc: RealPropName
  SurfaceGO_SetSpecularReflectionConc(o: GO, r: RealVal): Ok !
```

```
        replica_failure GO_PropError
  SurfaceGO_TransmissionCoeff: RealPropName
  SurfaceGO_SetTransmissionCoeff(o: GO, r: RealVal): Ok !
        replica_failure GO_PropError
  SurfaceGO_SpecularReflectionColor: ColorPropName
  SurfaceGO_SetSpecularReflectionColor(o: GO, color: ColorVal): Ok !
        replica_failure GO_PropError
  SurfaceGO_Lighting: BooleanPropName
  SurfaceGO_SetLighting(o: GO, t: BooleanVal): Ok ! replica_failure
        GO_PropError
  SurfaceGO_BackfaceCulling: BooleanPropName
  SurfaceGO_SetBackfaceCulling(o: GO, t: BooleanVal): Ok !
        replica_failure GO_PropError
  SurfaceGO_Shading: ShadingPropName
  SurfaceGO_SetShading(o: GO, sh: ShadingVal): Ok ! replica_failure
        GO_PropError
  SurfaceGO_EdgeVisibility: BooleanPropName
  SurfaceGO_SetEdgeVisibility(o: GO, b: BoolVal): Ok ! replica_failure
        GO_PropError
  SurfaceGO_EdgeColor: ColorPropName
  SurfaceGO_SetEdgeColor(o: GO, color: ColorVal): Ok ! replica_failure
        GO_PropError
  SurfaceGO_EdgeType: LineTypePropName
  SurfaceGO_SetEdgeType(o: GO, lt: LineTypeVal): Ok ! replica_failure
        GO_PropError
  SurfaceGO_EdgeWidth: RealPropName
  SurfaceGO_SetEdgeWidth(o: GO, r: RealVal): Ok ! replica_failure
        GO_PropError
  SurfaceGO_TexImg: TexImagePropName
  SurfaceGO_SetTexImg(o: GO, r: TexImage): Ok ! replica_failure
        GO_PropError
  SurfaceGO_TexRepeatS: BooleanPropName
  SurfaceGO_SetTexRepeatS(o: GO, r: Bool): Ok ! replica_failure
        GO_PropError
  SurfaceGO_TexRepeatT: BooleanPropName
  SurfaceGO_SetTexRepeatT(o: GO, r: Bool): Ok ! replica_failure
        GO_PropError
  SurfaceGO_TexOn: BooleanPropName
  SurfaceGO_SetTexOn(o: GO, r: Bool): Ok ! replica_failure GO_PropError
  SurfaceGO_TexModel: TexModelPropName
  SurfaceGO_SetTexModel(o: GO, r: TexModel): Ok ! replica_failure
        GO_PropError
  SurfaceGO_TexBlendColor: ColorPropName
  SurfaceGO_SetTexBlendColor(o: GO, r: Color): Ok ! replica_failure
        GO_PropError
TYPE
  SurfaceGO <: GO
  ColorVal = ColorPropVal + Color + Text
  BooleanVal = BooleanPropVal + Bool
  RealVal = RealPropVal + Real + Int
  LineTypeVal = LineTypePropVal + LineType
  RasterModeVal = RasterModePropVal + RasterMode
  ShadingVal = ShadingPropVal + Shading
```

H.1.25 Text2DGO

```
  Text2DGO_New(p: PointVal, t: StringVal, a: AlignmentVal): Text2DGO
  Text2DGO_NewWithSpacing(p: Pointval, t: StringVal, a: AlignmentVal,
                          s: RealVal): Text2DGO
  Text2DGO_GetScreenPosition(t: Text2DGO): [Int,Int]
  Text2DGO_GetScreenExtent(t: Text2DGO): [Int,Int]
  Text2DGO_IsVisible(t: Text2DGO): Bool
WHERE
  Text2DGO <: TextGO
```

H.1.26 TextGO

```
  TextGO_Position: PositionPropName
  TextGO_String: StringPropName
  TextGO_Alignment: TextAlignPropName
  TextGO_Spacing: RealPropName
  TextGO_SetPosition(go: GO, pos: PointVal): OK
  TextGO_SetString(go: GO, string: StringVal): Ok
  TextGO_SetAlignment(go: GO, align: AlignmentVal): Ok
  TextGO_SetSpacing(go: GO, radius: RealVal): Ok
  TextGO_FontFamily: FontFamilyPropName
  TextGO_SetFontFamily(go: GO, xf: FontFamilyVal): Ok
  TextGO_GetFontFamily(go: GO): FontFamilyVal ! GO_PropUndefined
  TextGO_FontStyle: FontStylePropName
  TextGO_SetFontStyle(go: GO, xf: FontStyleVal): Ok
  TextGO_GetFontStyle(go: GO): FontStyleVal ! GO_PropUndefined
  TextGO_FontSize: RealPropName
  TextGO_SetFontSize(go: GO, xf: RealVal): Ok
  TextGO_GetFontSize(go: GO): RealVal ! GO_PropUndefined
  TextGO_FontColor: ColorPropName
  TextGO_SetFontColor(go: GO, c: ColorVal): Ok
  TextGO_GetFontColor(go: GO): ColorVal ! GO_PropUndefined
WHERE
  TextGO <: GO
  PointVal = PointPropVal + Point3
  StringVal = StringPropVal + TEXT
  AlignmentVal = TEXT (One of "Left", "Right", "Center")
  RealVal = RealPropVal + Real + Int
  FontFamilyVal = StringPropVal + Text
  FontStyleVal = FontStylePropVal +
                 Text (one of "None", "Bold", "Italic")
  RealVal = RealPropVal + Real + Int
  ColorVal = ColorPropVal + Color + Text
```

H.1.27 TorusGO

```
   TorusGO_New(center normal: PointVal, rad1 rad2: RealVal): TorusGO
   TorusGO_NewWithPrec(c n: PointVal, r1 r2: RealVal, prec: Int):
               TorusGO
   TorusGO_Precision: IntPropName
   TorusGO_Center: PointPropName
   TorusGO_Normal: PointPropName
   TorusGO_Radius1: RealPropName
   TorusGO_Radius2: RealPropName
   TorusGO_SetCenter(go: GO, center: PointVal): Ok
   TorusGO_SetNormal(go: GO, normal: PointVal): Ok
   TorusGO_SetRadius1(go: GO, radius: RealVal): Ok
   TorusGO_SetRadius2(go: GO, radius: RealVal): Ok
   TorusGO_SetPrecision(go: GO, prec: IntVal): Ok
WHERE
   TorusGO <: SurfaceGO
   PointVal = PointPropVal + Point3
   RealVal = RealPropVal + Real + Int
   IntVal = IntPropVal + Int
```

H.1.28 VectorLightGO

```
   VectorLightGO_New(c: ColorVal, dir: PointVal, int: RealVal):
          VectorLightGO ! GO_PropError
   VectorLightGO_Direction: PointPropName
   VectorLightGO_SetDirection(l: VectorLightGO, dir: PointVal): Ok !
          replica_failure GO_PropError
WHERE
   VectorLightGO <: LightGO
   PointVal = PointPropVal + Point3
   ColorVal = ColorPropVal + Color + Text
   RealVal = RealPropVal + Real + Int
```

H.2 Properties

Properties are used to define the attributes of GOs. All properties inherit from the base Prop object, and use PropCB to be notified of changes.

H.2.1 Prop

```
   Prop_BadMethod: Exception
   Prop_BadInterval: Exception
```

```
TYPES
  Prop     <: ProxiedObj
  PropName <: ProxiedObj
  PropVal  <: ProxiedObj
  PropBeh  <: ProxiedObj
  PropRequest <: ProxiedObj & { start: () => Real, dur: () => Real }
```

H.2.2 PropCB

```
  PropCB_New(obj: Prop, overrides: Obj): T;
  PropCB_Cancel(cbobj: T): T;
WHERE
  T <: {simple} & overrides;
  overrides contains one or more of these callback methods:
       pre`init(obj: Prop, beh: PropBeh): bool;
       post`init(obj: Prop, beh: PropBeh): bool;
       pre`setBeh(obj: Prop, beh: PropBeh): bool;
       post`setBeh(obj: Prop, beh: PropBeh): bool;
       pre`anyChange(obj: Prop);
       post`anyChange(obj: Prop);

  Where Prop is a Property and PropBeh is a Property Behavior of the
  appropriate types
```

H.2.3 BooleanProp

```
  BooleanProp_NewConst(b: Bool): BooleanPropVal
  BooleanProp_NewSync(ah: AnimHandle, b: Bool): BooleanPropVal
  BooleanProp_NewAsync(beh: BooleanPropAsyncBeh): BooleanPropVal
  BooleanProp_NewDep(beh: BooleanPropDepBeh): BooleanPropVal
  BooleanProp_NewConstBeh(b: Bool): BooleanPropConstBeh
  BooleanProp_NewSyncBeh(ah: AnimHandle, b: Bool): BooleanPropSyncBeh
  BooleanProp_NewAsyncBeh(compute: M1):BooleanPropAsyncBeh
  BooleanProp_NewDepBeh(compute: M2):BooleanPropDepBeh
  BooleanProp_NewRequest(start dur: Num, value: M3): BooleanPropRequest
WHERE
  BooleanPropName <: PropName & { bind: (v: BooleanPropVal) => Prop }
  BooleanPropVal <: PropVal &
            { getBeh: () => BooleanPropBeh ! replica_failure,
              setBeh: (BooleanPropBeh) => Ok ! replica_failure,
              get: () => Bool ! replica_failure,
              value: (Num) => Bool ! replica_failure }
  BooleanPropBeh <: PropBeh
  BooleanPropConstBeh <: BooleanPropBeh & { set: (Bool) => Ok }
  BooleanPropSyncBeh <: BooleanPropBeh &
            { addRequest: (BooleanPropRequest) => Ok ! Prop_BadInterval,
              change: (Bool,Num) => Ok ! Prop_BadInterval }
  BooleanPropAsyncBeh <: BooleanPropBeh & { compute: M1 }
  BooleanPropDepBeh <: BooleanPropBeh & { compute: M2 }
```

```
  BooleanPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: BooleanPropAsyncBeh) (Real) => Bool
  M2 = Self (X <: BooleanPropDepBeh) (Real) => Bool
  M3 = Self (X <: BooleanPropRequest) (Bool,Real) => Bool
  Num = Real + Int
```

H.2.4 ColorProp

```
  ColorProp_NewConst(r: Col): ColorPropVal
  ColorProp_NewSync(ah: AnimHandle, r: Col): ColorPropVal
  ColorProp_NewAsync(beh: ColorPropAsyncBeh): ColorPropVal
  ColorProp_NewDep(beh: ColorPropDepBeh): ColorPropVal
  ColorProp_NewConstBeh(r: Col): ColorPropConstBeh
  ColorProp_NewSyncBeh(ah: AnimHandle, r: Col): ColorPropSyncBeh
  ColorProp_NewAsyncBeh(compute: M1):ColorPropAsyncBeh
  ColorProp_NewDepBeh(compute: M2):ColorPropDepBeh
  ColorProp_NewRequest(start dur: Num, value: M3): ColorPropRequest
WHERE
  ColorPropName <: PropName & { bind: (v: ColorPropVal) => Prop }
  ColorPropVal <: PropVal &
         { getBeh: () => ColorPropBeh ! replica_failure,
           setBeh: (ColorPropBeh) => Ok ! replica_failure,
           get: () => Color ! replica_failure,
           value: (Num) => Color ! replica_failure }
  ColorPropBeh <: PropBeh
  ColorPropConstBeh <: ColorPropBeh & { set: (Col) => Ok }
  ColorPropSyncBeh <: ColorPropBeh &
         { addRequest: (ColorPropRequest) => Ok ! Prop_BadInterval,
           rgbLinChangeTo: (Col,Num,Num) => Ok ! Prop_BadInterval }
  ColorPropAsyncBeh <: ColorPropBeh & { compute: M1 }
  ColorPropDepBeh <: ColorPropBeh & { compute: M2 }
  ColorPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: ColorPropAsyncBeh) (Real) => Color
  M2 = Self (X <: ColorPropDepBeh) (Real) => Color
  M3 = Self (X <: ColorPropRequest) (Color,Real) => Color
  Col = Color + Text
  Num = Real + Int
```

H.2.5 FontStyleProp

```
  FontStyleProp_NewConst(lt: FontStyle): FontStylePropVal
  FontStyleProp_NewSync(ah: AnimHandle, lt: FontStyle):
           FontStylePropVal
  FontStyleProp_NewAsync(beh: FontStylePropAsyncBeh): FontStylePropVal
  FontStyleProp_NewDep(beh: FontStylePropDepBeh): FontStylePropVal
  FontStyleProp_NewConstBeh(lt: FontStyle): FontStylePropConstBeh
  FontStyleProp_NewSyncBeh(ah: AnimHandle, lt: FontStyle):
           FontStylePropSyncBeh
  FontStyleProp_NewAsyncBeh(compute: M1):FontStylePropAsyncBeh
```

```
   FontStyleProp_NewDepBeh(compute: M2):FontStylePropDepBeh
   FontStyleProp_NewRequest(start dur: Num, value: M3):
            FontStylePropRequest
WHERE
   FontStylePropName <: PropName & { bind: (v: FontStylePropVal) => Prop
   FontStylePropVal <: PropVal & { getBeh: () => FontStylePropBeh,
                                   setBeh: (FontStylePropBeh) => Ok,
                                   get: () => FontStyle,
                                   value: (Num) => FontStyle }
   FontStylePropBeh <: PropBeh
   FontStylePropConstBeh <: FontStylePropBeh & {set: (FontStyle) => Ok }
   FontStylePropSyncBeh <: FontStylePropBeh &
          { addRequest: (FontStylePropRequest) => Ok ! Prop_BadInterval,
            change: (FontStyle,Num) => Ok ! Prop_BadInterval }
   FontStylePropAsyncBeh <: FontStylePropBeh & { compute: M1 }
   FontStylePropDepBeh <: FontStylePropBeh & { compute: M2 }
   FontStylePropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: FontStylePropAsyncBeh) (Real) => FontStyle
   M2 = Self (X <: FontStylePropDepBeh) (Real) => FontStyle
   M3 = Self (X <: FontStylePropRequest) (FontStyle,Real) => FontStyle
   FontStyle = Text  (one of "None","Bold","Italic", "BoldItalic")
   Num = Real + Int
```

H.2.6 IntProp

```
   IntProp_NewConst(r: Int): IntPropVal
   IntProp_NewSync(ah: AnimHandle, r: Int): IntPropVal
   IntProp_NewAsync(beh: IntPropAsyncBeh): IntPropVal
   IntProp_NewDep(beh: IntPropDepBeh): IntPropVal
   IntProp_NewConstBeh(r: Int): IntPropConstBeh
   IntProp_NewSyncBeh(ah: AnimHandle, r: Int): IntPropSyncBeh
   IntProp_NewAsyncBeh(compute: M1):IntPropAsyncBeh
   IntProp_NewDepBeh(compute: M2):IntPropDepBeh
   IntProp_NewRequest(start dur: Num, value: M3): IntPropRequest
WHERE
   IntPropName <: PropName & { bind: (v: IntPropVal) => Prop }
   IntPropVal <: PropVal & { getBeh: () => IntPropBeh ! replica_failure,
                    setBeh: (IntPropBeh) => Ok ! replica_failure,
                    get: () => Int ! replica_failure,
                    value: (Num) => Int ! replica_failure }
   IntPropBeh <: PropBeh
   IntPropConstBeh <: IntPropBeh & { set: (Int) => Ok }
   IntPropSyncBeh <: IntPropBeh &
             {  addRequest: (IntPropRequest) => Ok ! Prop_BadInterval,
                linChangeTo: (Int,Num,Num) => Ok ! Prop_BadInterval,
                linChangeBy: (Int,Num,Num) => Ok ! Prop_BadInterval }
   IntPropAsyncBeh <: IntPropBeh & { compute: M1 }
   IntPropDepBeh <: IntPropBeh & { compute: M2 }
   IntPropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: IntPropAsyncBeh) (Real) => Int
   M2 = Self (X <: RealPropDepBeh) (Real) => Int
   M3 = Self (X <: RealPropRequest) (Real,Real) => Int
```

```
  Num = Real + Int
```

H.2.7 LineTypeProp

```
  LineTypeProp_NewConst(lt: LineType): LineTypePropVal
  LineTypeProp_NewSync(ah: AnimHandle, lt: LineType): LineTypePropVal
  LineTypeProp_NewAsync(beh: LineTypePropAsyncBeh): LineTypePropVal
  LineTypeProp_NewDep(beh: LineTypePropDepBeh): LineTypePropVal
  LineTypeProp_NewConstBeh(lt: LineType): LineTypePropConstBeh
  LineTypeProp_NewSyncBeh(ah: AnimHandle, lt: LineType):
            LineTypePropSyncBeh
  LineTypeProp_NewAsyncBeh(compute: M1):LineTypePropAsyncBeh
  LineTypeProp_NewDepBeh(compute: M2):LineTypePropDepBeh
  LineTypeProp_NewRequest(start dur: Num, value: M3):
            LineTypePropRequest
WHERE
  LineTypePropName <: PropName & { bind: (v: LineTypePropVal) => Prop }
  LineTypePropVal <: PropVal &
         {  getBeh: () => LineTypePropBeh ! replica_failure,
            setBeh: (LineTypePropBeh) => Ok ! replica_failure,
            get: () => LineType ! replica_failure,
            value: (Num) => LineType ! replica_failure }
  LineTypePropBeh <: PropBeh
  LineTypePropConstBeh <: LineTypePropBeh & { set: (LineType) => Ok }
  LineTypePropSyncBeh <: LineTypePropBeh &
         {  addRequest: (LineTypePropRequest) => Ok ! Prop_BadInterval,
            change: (LineType,Num) => Ok ! Prop_BadInterval }
  LineTypePropAsyncBeh <: LineTypePropBeh & { compute: M1 }
  LineTypePropDepBeh <: LineTypePropBeh & { compute: M2 }
  LineTypePropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: LineTypePropAsyncBeh) (Real) => LineType
  M2 = Self (X <: LineTypePropDepBeh) (Real) => LineType
  M3 = Self (X <: LineTypePropRequest) (LineType,Real) => LineType
  LineType = Text  (one of "Solid", "Dashed", "Dotted", "DashDot")
  Num = Real + Int
```

H.2.8 MarkerTypeProp

```
  MarkerTypeProp_NewConst(lt: MarkerType): MarkerTypePropVal
  MarkerTypeProp_NewSync(ah: AnimHandle, lt: MarkerType):
              MarkerTypePropVal
  MarkerTypeProp_NewAsync(beh: MarkerTypePropAsyncBeh):
              MarkerTypePropVal
  MarkerTypeProp_NewDep(beh: MarkerTypePropDepBeh): MarkerTypePropVal
  MarkerTypeProp_NewConstBeh(lt: MarkerType): MarkerTypePropConstBeh
  MarkerTypeProp_NewSyncBeh(ah: AnimHandle,
                            t: MarkerType): MarkerTypePropSyncBeh
  MarkerTypeProp_NewAsyncBeh(compute: M1):MarkerTypePropAsyncBeh
  MarkerTypeProp_NewDepBeh(compute: M2):MarkerTypePropDepBeh
```

```
   MarkerTypeProp_NewRequest(start dur: Num, value: M3):
               MarkerTypePropRequest
WHERE
   MarkerTypePropName <: PropName &
                  { bind: (v: MarkerTypePropVal) => Prop }
   MarkerTypePropVal <: PropVal &
                  {  getBeh: () => MarkerTypePropBeh ! replica_failure,
                     setBeh: (MarkerTypePropBeh) => Ok ! replica_failure,
                     get: () => MarkerType ! replica_failure,
                     value: (Num) => MarkerType ! replica_failure }
   MarkerTypePropBeh <: PropBeh
   MarkerTypePropConstBeh <: MarkerTypePropBeh &
                  { set: (MarkerType) => Ok }
   MarkerTypePropSyncBeh <: MarkerTypePropBeh &
          { addRequest: (MarkerTypePropRequest) => Ok ! Prop_BadInterval,
           change: (MarkerType,Num) => Ok ! Prop_BadInterval }
   MarkerTypePropAsyncBeh <: MarkerTypePropBeh & { compute: M1 }
   MarkerTypePropDepBeh <: MarkerTypePropBeh & { compute: M2 }
   MarkerTypePropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: MarkerTypePropAsyncBeh) (Real) => MarkerType
   M2 = Self (X <: MarkerTypePropDepBeh) (Real) => MarkerType
   M3 = Self (X <: MarkerTypePropRequest) (MarkerType,Real) =>
                  MarkerType
   MarkerType = Text  (one of "Dot", "Circle", "Cross", "Asterisk", "X")
   Num = Real + Int
```

H.2.9 Point2Prop

```
   Point2Prop_NewConst(r: Point2): Point2PropVal
   Point2Prop_NewSync(ah: AnimHandle, r: Point2): Point2PropVal
   Point2Prop_NewAsync(beh: Point2PropAsyncBeh): Point2PropVal
   Point2Prop_NewDep(beh: Point2PropDepBeh): Point2PropVal
   Point2Prop_NewConstBeh(r: Point2): Point2PropConstBeh
   Point2Prop_NewSyncBeh(ah: AnimHandle, r: Point2): Point2PropSyncBeh
   Point2Prop_NewAsyncBeh(compute: M1):Point2PropAsyncBeh
   Point2Prop_NewDepBeh(compute: M2):Point2PropDepBeh
   Point2Prop_NewRequest(start dur: Num, value: M3): Point2PropRequest
WHERE
   Point2PropName <: PropName & { bind: (v: Point2PropVal) => Prop }
   Point2PropVal <: PropVal &
             {  getBeh: () => Point2PropBeh ! replica_failure,
                setBeh: (Point2PropBeh) => Ok ! replica_failure,
                get: () => Point2 ! replica_failure,
                value: (Num) => Point2 ! replica_failure }
   Point2PropBeh <: PropBeh
   Point2PropConstBeh <: Point2PropBeh & { set: (Point2) => Ok }
   Point2PropSyncBeh <: Point2PropBeh &
             {  addRequest: (Point2PropRequest) => Ok ! Prop_BadInterval,
                linMoveTo: (Point2,Num,Num) => Ok ! Prop_BadInterval,
                linMoveBy: (Point2,Num,Num) => Ok ! Prop_BadInterval }
   Point2PropAsyncBeh <: Point2PropBeh & { compute: M1 }
   Point2PropDepBeh <: Point2PropBeh & { compute: M2 }
   Point2PropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: Point2PropAsyncBeh) (Real) => Point2
```

```
  M2 = Self (X <: Point2PropDepBeh) (Real) => Point2
  M3 = Self (X <: Point2PropRequest) (Point2,Real) => Point2
  Num = Real + Int
```

H.2.10 PointProp

```
  PointProp_NewConst(r: Point3): PointPropVal
  PointProp_NewSync(ah: AnimHandle, r: Point3): PointPropVal
  PointProp_NewAsync(beh: PointPropAsyncBeh): PointPropVal
  PointProp_NewDep(beh: PointPropDepBeh): PointPropVal
  PointProp_NewConstBeh(r: Point3): PointPropConstBeh
  PointProp_NewSyncBeh(ah: AnimHandle, r: Point3): PointPropSyncBeh
  PointProp_NewAsyncBeh(compute: M1):PointPropAsyncBeh
  PointProp_NewDepBeh(compute: M2):PointPropDepBeh
  PointProp_NewRequest(start dur: Num, value: M3): PointPropRequest
WHERE
  PointPropName <: PropName & { bind: (v: PointPropVal) => Prop }
  PointPropVal <: PropVal &
            { getBeh: () => PointPropBeh ! replica_failure,
              setBeh: (PointPropBeh) => Ok ! replica_failure,
              get: () => Point3 ! replica_failure,
              value: (Num) => Point3 ! replica_failure }
  PointPropBeh <: PropBeh
  PointPropConstBeh <: PointPropBeh & { set: (Point3) => Ok }
  PointPropSyncBeh <: PointPropBeh &
            { addRequest: (PointPropRequest) => Ok ! Prop_BadInterval,
              linMoveTo: (Point3,Num,Num) => Ok ! Prop_BadInterval,
              linMoveBy: (Point3,Num,Num) => Ok ! Prop_BadInterval }
  PointPropAsyncBeh <: PointPropBeh & { compute: M1 }
  PointPropDepBeh <: PointPropBeh & { compute: M2 }
  PointPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: PointPropAsyncBeh) (Real) => Point3
  M2 = Self (X <: PointPropDepBeh) (Real) => Point3
  M3 = Self (X <: PointPropRequest) (Point3,Real) => Point3
  Num = Real + Int
```

H.2.11 RasterModeProp

```
  RasterModeProp_NewConst(lt: RasterMode): RasterModePropVal
  RasterModeProp_NewSync(ah: AnimHandle, lt: RasterMode):
         RasterModePropVal
  RasterModeProp_NewAsync(beh: RasterModePropAsyncBeh):
         RasterModePropVal
  RasterModeProp_NewDep(beh: RasterModePropDepBeh): RasterModePropVal
  RasterModeProp_NewConstBeh(lt: RasterMode): RasterModePropConstBeh
  RasterModeProp_NewSyncBeh(ah: AnimHandle,
                            lt: RasterMode): RasterModePropSyncBeh
  RasterModeProp_NewAsyncBeh(compute: M1):RasterModePropAsyncBeh
  RasterModeProp_NewDepBeh(compute: M2):RasterModePropDepBeh
```

```
   RasterModeProp_NewRequest(start dur: Num, value: M3):
          RasterModePropRequest
WHERE
   RasterModePropName <: PropName &
             { bind: (v: RasterModePropVal) => Prop }
   RasterModePropVal <: PropVal &
             { getBeh: () => RasterModePropBeh ! replica_failure,
               setBeh: (RasterModePropBeh) => Ok ! replica_failure,
               get: () => RasterMode ! replica_failure,
               value: (Num) => RasterMode ! replica_failure }
   RasterModePropBeh <: PropBeh
   RasterModePropConstBeh <: RasterModePropBeh &
             { set: (RasterMode) => Ok }
   RasterModePropSyncBeh <: RasterModePropBeh &
        { addRequest: (RasterModePropRequest) => Ok ! Prop_BadInterval,
          change: (RasterMode,Num) => Ok ! Prop_BadInterval }
   RasterModePropAsyncBeh <: RasterModePropBeh & { compute: M1 }
   RasterModePropDepBeh <: RasterModePropBeh & { compute: M2 }
   RasterModePropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: RasterModePropAsyncBeh) (Real) => RasterMode
   M2 = Self (X <: RasterModePropDepBeh) (Real) => RasterMode
   M3 = Self (X <: RasterModePropRequest) (RasterMode,Real) =>
               RasterMode
   RasterMode = Text  (one of "Vector" "Hollow", "Solid", "Empty")
   Num = Real + Int
```

H.2.12 RealProp

```
   RealProp_NewConst(r: Num): RealPropVal
   RealProp_NewSync(ah: AnimHandle, r: Num): RealPropVal
   RealProp_NewAsync(beh: RealPropAsyncBeh): RealPropVal
   RealProp_NewDep(beh: RealPropDepBeh): RealPropVal
   RealProp_NewConstBeh(r: Num): RealPropConstBeh
   RealProp_NewSyncBeh(ah: AnimHandle, r: Num): RealPropSyncBeh
   RealProp_NewAsyncBeh(compute: M1):RealPropAsyncBeh
   RealProp_NewDepBeh(compute: M2):RealPropDepBeh
   RealProp_NewRequest(start dur: Num, value: M3): RealPropRequest
WHERE
   RealPropName <: PropName & { bind: (v: RealPropVal) => Prop }
   RealPropVal <: PropVal &
        { getBeh: () => RealPropBeh ! replica_failure,
          setBeh: (RealPropBeh) => Ok ! replica_failure,
          get: () => Real ! replica_failure,
          value: (Num) => Real ! replica_failure }
   RealPropBeh <: PropBeh
   RealPropConstBeh <: RealPropBeh & { set: (Num) => Ok }
   RealPropSyncBeh <: RealPropBeh &
          { addRequest: (RealPropRequest) => Ok ! Prop_BadInterval,
            linChangeTo: (Num,Num,Num) => Ok ! Prop_BadInterval,
            linChangeBy: (Num,Num,Num) => Ok ! Prop_BadInterval }
   RealPropAsyncBeh <: RealPropBeh & { compute: M1 }
   RealPropDepBeh <: RealPropBeh & { compute: M2 }
   RealPropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: RealPropAsyncBeh) (Real) => Real
```

```
  M2 = Self (X <: RealPropDepBeh) (Real) => Real
  M3 = Self (X <: RealPropRequest) (Real,Real) => Real
  Num = Real + Int
```

H.2.13 ShadingProp

```
  ShadingProp_NewConst(lt: Shading): ShadingPropVal
  ShadingProp_NewSync(ah: AnimHandle, lt: Shading): ShadingPropVal
  ShadingProp_NewAsync(beh: ShadingPropAsyncBeh): ShadingPropVal
  ShadingProp_NewDep(beh: ShadingPropDepBeh): ShadingPropVal
  ShadingProp_NewConstBeh(lt: Shading): ShadingPropConstBeh
  ShadingProp_NewSyncBeh(ah: AnimHandle, lt: Shading):
           ShadingPropSyncBeh
  ShadingProp_NewAsyncBeh(compute: M1):ShadingPropAsyncBeh
  ShadingProp_NewDepBeh(compute: M2):ShadingPropDepBeh
  ShadingProp_NewRequest(start dur: Num, value: M3): ShadingPropRequest
WHERE
  ShadingPropName <: PropName & { bind: (v: ShadingPropVal) => Prop }
  ShadingPropVal <: PropVal &
           {  getBeh: () => ShadingPropBeh ! replica_failure,
              setBeh: (ShadingPropBeh) => Ok ! replica_failure,
              get: () => Shading ! replica_failure,
              value: (Num) => Shading ! replica_failure }
  ShadingPropBeh <: PropBeh
  ShadingPropConstBeh <: ShadingPropBeh & { set: (Shading) => Ok }
  ShadingPropSyncBeh <: ShadingPropBeh &
         {  addRequest: (ShadingPropRequest) => Ok ! Prop_BadInterval,
            change: (Shading,Num) => Ok ! Prop_BadInterval }
  ShadingPropAsyncBeh <: ShadingPropBeh & { compute : M1 }
  ShadingPropDepBeh <: ShadingPropBeh & { compute: M2 }
  ShadingPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: ShadingPropAsyncBeh) (Real) => Shading
  M2 = Self (X <: ShadingPropDepBeh) (Real) => Shading
  M3 = Self (X <: ShadingPropRequest) (Shading,Real) => Shading
  Shading = Text  ("Flat" or "Gouraud")
  Num = Real + Int
```

H.2.14 StringProp

```
  StringProp_NewConst(t: TEXT): StringPropVal
  StringProp_NewSync(ah: AnimHandle, t: TEXT): StringPropVal
  StringProp_NewAsync(beh: StringPropAsyncBeh): StringPropVal
  StringProp_NewDep(beh: StringPropDepBeh): StringPropVal
  StringProp_NewConstBeh(t: TEXT): StringPropConstBeh
  StringProp_NewSyncBeh(ah: AnimHandle, t: TEXT): StringPropSyncBeh
  StringProp_NewAsyncBeh(compute: M1):StringPropAsyncBeh
  StringProp_NewDepBeh(compute: M2):StringPropDepBeh
  StringProp_NewRequest(start dur: Num, value: M3): StringPropRequest
WHERE
```

```
  StringPropName <: PropName & { bind: (v: StringPropVal) => Prop }
  StringPropVal <: PropVal & { getBeh: () => StringPropBeh,
                               setBeh: (StringPropBeh) => Ok,
                               get: () => TEXT,
                               value: (Num) => TEXT }
  StringPropBeh <: PropBeh
  StringPropConstBeh <: StringPropBeh & { set: (TEXT) => Ok }
  StringPropSyncBeh <: StringPropBeh &
           { addRequest: (StringPropRequest) => Ok ! Prop_BadInterval,
             change: (TEXT,Num) => Ok ! Prop_BadInterval }
  StringPropAsyncBeh <: StringPropBeh & { compute: M1 }
  StringPropDepBeh <: StringPropBeh & { compute: M2 }
  StringPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: StringPropAsyncBeh) (Real) => TEXT
  M2 = Self (X <: StringPropDepBeh) (Real) => TEXT
  M3 = Self (X <: StringPropRequest) (TEXT,Real) => TEXT
  Num = Real + Int
```

H.2.15 TexImageProp

```
  TexImageProp_NewConst(m: TexImage): TexImagePropVal
  TexImageProp_NewSync(ah: AnimHandle, m: TexImage): TexImagePropVal
  TexImageProp_NewAsync(beh: TexImagePropAsyncBeh): TexImagePropVal
  TexImageProp_NewDep(beh: TexImagePropDepBeh): TexImagePropVal
  TexImageProp_NewConstBeh(m: TexImage): TexImagePropConstBeh
  TexImageProp_NewSyncBeh(ah: AnimHandle, m: TexImage):
           TexImagePropSyncBeh
  TexImageProp_NewAsyncBeh(compute: M1):TexImagePropAsyncBeh
  TexImageProp_NewDepBeh(compute: M2):TexImagePropDepBeh
  TexImageProp_NewRequest(start dur: Num, value: M3):
           TexImagePropRequest
WHERE
  TexImagePropName <: PropName & { bind: (v: TexImagePropVal) => Prop }
  TexImagePropVal <: PropVal & { getBeh: () => TexImagePropBeh,
                                 setBeh: (TexImagePropBeh) => Ok,
                                 get: () => TexImage,
                                 value: (Num) => TexImage }
  TexImagePropBeh <: PropBeh
  TexImagePropConstBeh <: TexImagePropBeh &
                                 { set: (TexImage) => Ok}
  TexImagePropSyncBeh <: TexImagePropBeh &
         { addRequest: (TexImagePropRequest) => Ok ! Prop_BadInterval}
  TexImagePropAsyncBeh <: TexImagePropBeh & { compute: M1 }
  TexImagePropDepBeh <: TexImagePropBeh & { compute: M2 }
  TexImagePropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: TexImagePropAsyncBeh) (Real) => TexImage
  M2 = Self (X <: TexImagePropDepBeh) (Real) => TexImage
  M3 = Self (X <: TexImagePropRequest) (TexImage,Real) => TexImage
  Num = Real = Int
```

H.2.16 TexModelProp

```
   TexModelProp_NewConst(lt: TexModel): TexModelPropVal
   TexModelProp_NewSync(ah: AnimHandle, lt: TexModel): TexModelPropVal
   TexModelProp_NewAsync(beh: TexModelPropAsyncBeh): TexModelPropVal
   TexModelProp_NewDep(beh: TexModelPropDepBeh): TexModelPropVal
   TexModelProp_NewConstBeh(lt: TexModel): TexModelPropConstBeh
   TexModelProp_NewSyncBeh(ah: AnimHandle, lt: TexModel):
          TexModelPropSyncBeh
   TexModelProp_NewAsyncBeh(compute: M1):TexModelPropAsyncBeh
   TexModelProp_NewDepBeh(compute: M2):TexModelPropDepBeh
   TexModelProp_NewRequest(start dur: Num, value: M3):
          TexModelPropRequest
WHERE
   TexModelPropName <: PropName & { bind: (v: TexModelPropVal) => Prop }
   TexModelPropVal <: PropVal & { getBeh: () => TexModelPropBeh,
                                  setBeh: (TexModelPropBeh) => Ok,
                                  get: () => TexModel,
                                  value: (Num) => TexModel }
   TexModelPropBeh <: PropBeh
   TexModelPropConstBeh <: TexModelPropBeh & { set: (TexModel) => Ok }
   TexModelPropSyncBeh <: TexModelPropBeh &
          { addRequest: (TexModelPropRequest) => Ok ! Prop_BadInterval,
            change: (TexModel,Num) => Ok ! Prop_BadInterval }
   TexModelPropAsyncBeh <: TexModelPropBeh & { compute: M1 }
   TexModelPropDepBeh <: TexModelPropBeh & { compute: M2 }
   TexModelPropRequest <: PropRequest & { value: M3 }
   M1 = Self (X <: TexModelPropAsyncBeh) (Real) => TexModel
   M2 = Self (X <: TexModelPropDepBeh) (Real) => TexModel
   M3 = Self (X <: TexModelPropRequest) (TexModel,Real) => TexModel
   TexModel = Text  (one of "Modulate", "Decl", "Blend")
   Num = Real + Int
```

H.2.17 TextAlignProp

```
   TextAlignProp_NewConst(lt: TextAlign): TextAlignPropVal
   TextAlignProp_NewSync(ah: AnimHandle, lt: TextAlign):
               TextAlignPropVal
   TextAlignProp_NewAsync(beh: TextAlignPropAsyncBeh): TextAlignPropVal
   TextAlignProp_NewDep(beh: TextAlignPropDepBeh): TextAlignPropVal
   TextAlignProp_NewConstBeh(lt: TextAlign): TextAlignPropConstBeh
   TextAlignProp_NewSyncBeh(ah: AnimHandle, lt: TextAlign):
               TextAlignPropSyncBeh
   TextAlignProp_NewAsyncBeh(compute: M1):TextAlignPropAsyncBeh
   TextAlignProp_NewDepBeh(compute: M2):TextAlignPropDepBeh
   TextAlignProp_NewRequest(start dur: Num, value: M3):
               TextAlignPropRequest
WHERE
   TextAlignPropName <: PropName &
             { bind: (v: TextAlignPropVal) => Prop }
   TextAlignPropVal <: PropVal & { getBeh: () => TextAlignPropBeh,
```

```
                                  setBeh: (TextAlignPropBeh) => Ok,
                                  get: () => TextAlign,
                                  value: (Num) => TextAlign }
  TextAlignPropBeh <: PropBeh
  TextAlignPropConstBeh <: TextAlignPropBeh & { set: (TextAlign) => Ok}
  TextAlignPropSyncBeh <: TextAlignPropBeh &
         { addRequest: (TextAlignPropRequest) => Ok ! Prop_BadInterval,
           change: (TextAlign,Num) => Ok ! Prop_BadInterval }
  TextAlignPropAsyncBeh <: TextAlignPropBeh & { compute: M1 }
  TextAlignPropDepBeh <: TextAlignPropBeh & { compute: M2 }
  TextAlignPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: TextAlignPropAsyncBeh) (Real) => TextAlign
  M2 = Self (X <: TextAlignPropDepBeh) (Real) => TextAlign
  M3 = Self (X <: TextAlignPropRequest) (TextAlign,Real) => TextAlign
  TextAlign = Text  (one of "Left", "Right", "Center")
  Num = Real + Int
```

H.2.18 TransformProp

```
  TransformProp_NewConst(m: Matrix4): TransformPropVal
  TransformProp_NewSync(ah: AnimHandle, m: Matrix4): TransformPropVal
  TransformProp_NewAsync(beh: TransformPropAsyncBeh): TransformPropVal
  TransformProp_NewDep(beh: TransformPropDepBeh): TransformPropVal
  TransformProp_NewConstBeh(m: Matrix4): TransformPropConstBeh
  TransformProp_NewSyncBeh(ah: AnimHandle, m: Matrix4):
            TransformPropSyncBeh
  TransformProp_NewAsyncBeh(compute: M1):TransformPropAsyncBeh
  TransformProp_NewDepBeh(compute: M2):TransformPropDepBeh
  TransformProp_NewRequest(start dur: Num, value: M3):
            TransformPropRequest
WHERE
  TransformPropName <: PropName &
            { bind: (v: TransformPropVal) => Prop }
  TransformPropVal <: PropVal &
            { getBeh: () => TransformPropBeh ! replica_failure,
              setBeh: (TransformPropBeh) => Ok ! replica_failure,
              get: () => Matrix4 ! replica_failure,
              value: (Num) => Matrix4 ! replica_failure }
  TransformPropBeh <: PropBeh
  TransformPropConstBeh <: TransformPropBeh &
                                  { set: (Matrix4) => Ok,
                                  compose: (Matrix4) => Ok,
                                  reset: () => Ok,
                                  translate: (Num,Num,Num) => Ok,
                                  scale: (Num,Num,Num) => Ok,
                                  rotateX: (Num) => Ok,
                                  rotateY: (Num) => Ok,
                                  rotateZ: (Num) => Ok }
  TransformPropSyncBeh <: TransformPropBeh &
       { addRequest: (TransformPropRequest) => Ok ! Prop_BadInterval,
         reset: (Num) => Ok ! Prop_BadInterval,
         changeTo: (Matrix4,Num,Num) => Ok ! Prop_BadInterval,
         translate: (Num,Num,Num,Num,Num) => Ok ! Prop_BadInterval,
         scale: (Num,Num,Num,Num,Num) => Ok ! Prop_BadInterval,
```

```
        rotateX: (Num,Num,Num) => Ok ! Prop_BadInterval,
        rotateY: (Num,Num,Num) => Ok ! Prop_BadInterval,
        rotateZ: (Num,Num,Num) => Ok ! Prop_BadInterval }
  TransformPropAsyncBeh <: TransformPropBeh & { compute: M1 }
  TransformPropDepBeh <: TransformPropBeh & { compute: M2 }
  TransformPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: TransformPropAsyncBeh) (Real) => Matrix4
  M2 = Self (X <: TransformPropDepBeh) (Real) => Matrix4
  M3 = Self (X <: TransformPropRequest) (Matrix4,Real) => Matrix4
  Num = Real = Int
```

H.2.19 TransmissionPatternProp

```
  TransmissionPatternProp_NewConst(lt: TransmissionPattern):
        TransmissionPatternPropVal
  TransmissionPatternProp_NewSync(ah: AnimHandle,
        lt: TransmissionPattern): TransmissionPatternPropVal
  TransmissionPatternProp_NewAsync(
     beh:TransmissionPatternPropAsyncBeh):TransmissionPatternPropVal
  TransmissionPatternProp_NewDep(beh: TransmissionPatternPropDepBeh):
        TransmissionPatternPropVal
  TransmissionPatternProp_NewConstBeh(lt: TransmissionPattern):
        TransmissionPatternPropConstBeh
  TransmissionPatternProp_NewSyncBeh(ah: AnimHandle,
        lt: TransmissionPattern): TransmissionPatternPropSyncBeh
  TransmissionPatternProp_NewAsyncBeh(compute: M1):
        TransmissionPatternPropAsyncBeh
  TransmissionPatternProp_NewDepBeh(compute: M2):
        TransmissionPatternPropDepBeh
  TransmissionPatternProp_NewRequest(start dur: Num, value: M3):
        TransmissionPatternPropRequest
WHERE
  TransmissionPatternPropName <: PropName &
        { bind: (v: TransmissionPatternPropVal) => Prop }
  TransmissionPatternPropVal <: PropVal &
     { getBeh: () => TransmissionPatternPropBeh ! replica_failure,
        setBeh: (TransmissionPatternPropBeh) => Ok ! replica_failure,
        get: () => TransmissionPattern ! replica_failure,
        value: (Num) => TransmissionPattern ! replica_failure }
  TransmissionPatternPropBeh <: PropBeh
  TransmissionPatternPropConstBeh <: TransmissionPatternPropBeh &
        { set: (TransmissionPattern) => Ok }
  TransmissionPatternPropSyncBeh <: TransmissionPatternPropBeh &
        { addRequest: (TransmissionPatternPropRequest) => Ok !
                    Prop_BadInterval,
          change: (TransmissionPattern,Num) => Ok ! Prop_BadInterval }
  TransmissionPatternPropAsyncBeh <: TransmissionPatternPropBeh &
        { compute: M1 }
  TransmissionPatternPropDepBeh <: TransmissionPatternPropBeh &
        { compute: M2 }
  TransmissionPatternPropRequest <: PropRequest & { value: M3 }
  M1 = Self (X <: TransmissionPatternPropAsyncBeh) (Real) =>
                TransmissionPattern
  M2 = Self (X <: TransmissionPatternPropDepBeh) (Real) =>
```

```
                TransmissionPattern
  M3 = Self (X <: TransmissionPatternPropRequest)
                (TransmissionPattern,Real) => TransmissionPattern
  TransmissionPattern = Text  (one of "Blending", "Stipple")
  Num = Real + Int
```

H.3 Animation Handles

H.3.1 AnimHandle

```
  AnimHandle_New(): AnimHandle
WHERE
  AnimHandle <: ProxiedObj & { animate: () => Ok ! replica_failure,
                  startAnimation: () => Ok ! replica_failure,
                  finishAnimation: () => Ok ! replica_failure,
                  stopAnimation: () => Ok ! replica_failure,
                  pauseAnimation: () => Ok ! replica_failure,
                  continueAnimation: () => Ok ! replica_failure,
                  getAnimationTime:() => Real ! replica_failure,
                  getAnimationLength:() => Real ! replica_failure,
                  goToAnimationTime:(Real) => Ok! replica_failure}
```

H.3.2 AnimHandleCB

```
  AnimHandleCB_New(obj: AnimHandle_T, overrides: Obj): T;
  AnimHandleCB_Cancel(cbobj: T): T;
WHERE
  T <: {simple} & overrides;
  overrides contains one or more of these callback methods:
       pre`anyChange(obj: AnimHandle_T);
       post`anyChange(obj: AnimHandle_T);
       pre`init(obj: AnimHandle_T): bool;
       post`init(obj: AnimHandle_T): bool;
       pre`startAnimation(obj: AnimHandle_T): bool;
       post`startAnimation(obj: AnimHandle_T): bool;
       pre`stopAnimation(obj: AnimHandle_T): bool;
       post`stopAnimation(obj: AnimHandle_T): bool;
       pre`pauseAnimation(obj: AnimHandle_T): bool;
       post`pauseAnimation(obj: AnimHandle_T): bool;
       pre`continueAnimation(obj: AnimHandle_T): bool;
       post`continueAnimation(obj: AnimHandle_T): bool;
       pre`goToAnimationTime(obj: AnimHandle_T, time: Real): bool;
       post`goToAnimationTime(obj: AnimHandle_T, time: Real): bool;
```

H.4 Interaction Callbacks

Interaction callbacks are used to obtain input from the user via a RootGO.

H.4.1 KeyCB

```
   KeyCB_New(invoke: M): KeyCB
WHERE
   KeyCB <: ProxiedObj & { invoke: M }
   M = Self (X <: KeyCB) (KeyRec) => Ok
   KeyRec = { change: Text, wentDown: Bool, modifiers: [Modifier] }
   Modifier = Text (one of "Left", "Middle", "Right",
                    "Shift", "Lock", "Control", "Option")
```

A KeyCB is used to obtain keystroke input.

H.4.2 MouseCB

```
   MouseCB_New(invoke: M): MouseCB
WHERE
   MouseCB <: ProxiedObj & { invoke: M }
   M = Self (X <: MouseCB) (MouseRec) => Ok
   MouseRec = { pos: Point2, change: Button,
                  modifiers: [Modifier], clickType: ClickType }
   Point2 = [2*Int]
   Button = Text     (one of "Left", "Middle", "Right")
   Modifier = Text   (a Button or one of "Shift", "Lock", "Control",
                      "Option")
   ClickType = Text  (one of "FirstDown", "OtherDown", "OtherUp",
                      "LastUp")
```

A MouseCB is used to obtain mouse button presses and releases.

H.4.3 PositionCB

```
   PositionCB_New(invoke: M): PositionCB
WHERE
   PositionCB <: ProxiedObj & { invoke: M }
   M = Self (X <: PositionCB) (PositionRec) => Ok
   PositionRec = { pos: Point2, modifiers: [Modifier] }
   Point2 = [2*Int]
   Modifier = Text (one of "Left", "Middle", "Right",
                    "Shift", "Lock", "Control", "Option")
```

A PositionCB is used to obtain mouse motion input.

H.5 Location Callbacks

H.5.1 ProjectionCB

```
   ProjectionCB_New(invoke: M): ProjectionCB
WHERE
   ProjectionCB <: ProxiedObj & { invoke: M }
   M = Self (X <: ProjectionCB) (ProjectionRec) => Ok
   ProjectionRec = { base => GraphicsBase,
                     point => [Point3,Point3,Point3],
                     relPos => [RelPosition,RelPosition,RelPosition]}
        where the 3 elements in point and relPos correspond to the
        Left, Right and Monocular viewpoint
   RelPosition = {"Front", "On", "Behind"}
```

A ProjectionCB is used to obtain the 2D projection of a 3D point in the scene.

H.5.2 TransformCB

```
   TransformCB_New(invoke: M): TransformCB
WHERE
   TransformCB <: ProxiedObj & { invoke: M }
   M = Self (X <: TransformCB) (TransformRec) => Ok
   TransformRec = { toWorld => Matrix4,
                    localOriginToWorld => Point3,
                    fromWorld => Matrix4,
                    worldOriginToLocal => Point3 }
```

A TransformCB is used to obtain the 3D transformations to and from the world coordinate system of a 3D point in the scene.

H.6 Graphics Bases

Graphics Bases specify which rendering subsystem is to be used to render a 3D scene into a RootGO. Not all Graphics Bases are available on all machines.

H.6.1 GraphicsBase

```
  GraphicsBase_Failure: Exception
WHERE
    GraphicsBase <: ProxiedObj &
           { windowPosSize: () => PosSizeRec,
             changeTitle: (Text) => Ok,
             awaitDelete: () => Ok,
             destroy: () => Ok }
    PosSizeRec = {origin, viewPortOrigin, viewPortDimen: Point}
```

GraphicsBase is the abstract base that all others inherit from.

H.6.2 Win_OpenGL_Base

```
  Win`OpenGL`Base_New(title: Text, x y w h: Int): Win`OpenGL`Base !
         GraphicsBase_Failure
  Win`OpenGL`Base_NewInWindow(title: Text, x y w h: Int,
                              win class: Text):
         Win`OpenGL`Base ! GraphicsBase_Failure
  Win`OpenGL`Base_NewStd(): Win`OpenGL`Base ! GraphicsBase_Failure
WHERE
  Win`OpenGL`Base <: GraphicsBase & {toggleFullScreen: () => Ok}
```

Render using OpenGL on the Windows platform.

H.6.3 Win_RW_Base

```
  Win`RW`Base_New(title: Text, x y w h: Int): Win`RW`Base !
         GraphicsBase_Failure
  Win`RW`Base_NewInWindow(title: Text, x y w h: Int, name class: Text):
         Win`RW`Base ! GraphicsBase_Failure
  Win`RW`Base_NewStd(): Win`RW`Base ! GraphicsBase_Failure
WHERE
  Win`RW`Base <: GraphicsBase & { toggleFullScreen: () => Ok}
```

Render using Renderware on the Windows platform.

H.6.4 X_OpenGL_Base

```
  X`OpenGL`Base_New(title: Text, x y w h: Int): X`OpenGL`Base !
         GraphicsBase_Failure
```

```
  X`OpenGL`Base_NewWithDisplay(title: Text, x y w h: Int,
         dpyName: Text): X`OpenGL`Base ! GraphicsBase_Failure
  X`OpenGL`Base_NewOnRoot(title: Text, x y w h: Int): X`OpenGL`Base !
         GraphicsBase_Failure
  X`OpenGL`Base_NewOnRootWithDisplay(title: Text, x y w h: Int,
         dpyName: Text): X`OpenGL`Base ! GraphicsBase_Failure
  X`OpenGL`Base_NewInWindow(title: Text, x y w h winID: Int):
         X`OpenGL`Base ! GraphicsBase_Failure
  X`OpenGL`Base_NewInWindowWithDisplay(title: Text, x y w h winID: Int,
         dpyName: Text): X`OpenGL`Base ! GraphicsBase_Failure
  X`OpenGL`Base_NewStd(): X`OpenGL`Base ! GraphicsBase_Failure
WHERE
  X`OpenGL`Base <: GraphicsBase
```

Render using OpenGL on the Unix/X11 platform.

H.7 Miscellaneous

H.7.1 Anim3D

```
  Anim3D_lock: Mutex
  Anim3D_now: Real
    The current value of the animation clock.
  Anim3D_ChangeClock(proc: ()->Real): Ok
    Change the animation clock.  The procedure "proc" is the new time
    function that returns the "current time".
  Anim3D_DefaultClock(): Ok
    Revert to the default, real-time animation clock.
  Anim3D_SetErrorWr(wr: Wr): Ok
    Set the writer to which animation server error messages will be
    written to be "wr". By default, error messages are written to
    "wr_stderr".
```

H.7.2 AnimHook

```
  AnimHook_AddBeforeHook(f: (Real)->Ok): Ok
  AnimHook_RemoveBeforeHook(f: (Real)->Ok): ((Real)->Ok | Ok)
  AnimHook_AddAfterHook(f: (Real)->Ok): Ok
  AnimHook_RemoveAfterHook(f: (Real)->Ok): ((Real)->Ok | Ok)
```

H.7.3 ProxiedObj

```
TYPE ProxiedObj <: { extend: Self(X) All(Y<:{simple}) (Y) => X & Y }
```

```
            Objects of this type also contain a field "raw",
            which is for internal use only. All objects must
            be simple.
```

H.7.4 TessSphere

```
  TessSphere_NewOmniGO(prec: int): GO;
```

H.7.5 TexImage

```
  TexImage_Error: Exception
  TexImage_New(fileName: Text): TexImage ! Thread.Alerted, Error
WHERE
  TexImage is opaque
```

APPENDIX I The Animation Time Module

In Section 5.7, we discuss our solution to clock synchronization across multiple machines. Since we cannot assume that all machines have their time clocks synchronized, we wrote a simple module to keep the clocks of our distributed processes synchronized. This appendix contains the code for that module.

To use the module, one process is chosen as a server and runs `animtime_serve(hostname)`, where `hostname` is the host to which the server network object should be exported to; the server object is the one line object embedded in the `net_export` statement, that contains a `get()` method to return the time on the server process. Any process can elect to be a client process. The client forks a thread, and uses a simple protocol to determine the time difference between the two processes. First, the client calls the server's `get()` method ten times, with a small pause between each invocation. By assuming that the round trip time for the call is symmetric (i.e., the delay is equal for sending the method to the remote host, and returning the time value), half the delay is subtracted from the server time, giving an approximation of the server's time when the method was invoked. The difference between the server's time and the local time, averaged over the ten invocations, is used to adjust the local time to correspond roughly to the server time.

This protocol is not extremely robust, as variations in network delay (both within a single call, and across multiple calls) exist and add noise to the system. However, we have found that it works reasonably well in practice, especially on local area networks.

I.1 animtime.obl

```
module animtime;

let serve = proc (host)
     net_export("timeserver", host,
               {get => meth(s) Anim3D_now end});
     ok;
   end;

let client = proc (host)
   var timeserver = ok;

   var offset = 0.0;

   var th = ok;
   var a = array_new(10, ok);
   var b = array_new(10, ok);
   var scan = true;
   var stop = true;

   let start = proc () thread_fork (proc ()
     loop
       try
         if scan then
           if timeserver is ok then
             timeserver := net_import("timeserver",host);
           end;
           scan := false;
           for i = 0 to 9 do
             lock Anim3D_lock do
               let t1 = sys_timeNow,
               t2 = timeserver.get(),
               t3 = sys_timeNow;
               a[i] := (t2 - t1) - ((t3-t1)/2.0);
             end;
             thread_pause(0.1);
             end;

           offset := a[0];
           for i = 1 to 9 do
             b[i] := ((a[i] - a[0])/10.0);
             offset := offset + b[i];
           end;
         end;
       except net_failure =>
         sys_printText("Timeserver.get() failed. " &
                       "Will try later.\n");
         sys_printFlush();
         timeserver := ok,
       thread_alerted =>
         sys_printText("Timeserver.get() interrupted. " &
                       "Will try later.\n");
         sys_printFlush();
       end;
       (* pause for an hour *)
       try
         thread_alertPause (3600.0);
```

```
        except thread_alerted => end;
          if stop then exit end;
        end;
        timeserver := ok;
      end, 20000) end;
      let ret = { stop => meth (s)
                            if th isnot ok then
                              stop := true;
                              thread_alert(th);
                              thread_join(th);
                              th := ok;
                              Anim3D_DefaultClock();
                            end;
                            ok;
                          end,
                  set => meth (s)
                            if th isnot ok then
                              scan := true;
                              thread_alert(th);
                            end;
                          end,
                  offset => meth(s) offset end,
                  start => meth (s)
                            if th is ok then
                              th := start();
                              thread_pause(1.0);
                              Anim3D_ChangeClock(proc ()
                                   sys_timeNow + offset;
                                end);
                            end;
                            ok;
                          end};
      ret.start();
      ret;
    end;

end module;

addhelp animtime short "The animation time synchronization package" full
"  animtime_serve(host: Text): Ok ! net_failure thread_alerted
      Export an animation time server to \"host\".  Other processes can
      import this to synchronize their animation clocks to us.
    animtime_client(host: Text): TimeClient ! net_failure thread_alerted
      Import an animation time server client.
TYPE
    TimeClient <: {start: () => Ok ! net_failure,
                   stop: () => Ok ! net_failure};
";
```