Collaborative Visualization using Facet Trees: Design and Implementation of a Prototype System

by
Aguido Horatio Davis, BInfTech, BSc(AMS).

Submitted in partial fulfilment of the requirements of the degree of Bachelor of Information Technology with Honours.

School of Computing and Information Technology,
Faculty of Engineering and Information Technology,
Griffith University, Queensland.

October, 2000.

Abstract

The computational science and engineering process (build a model from reality, run the model, analyze the results for prediction and insight) is increasingly being applied to larger and more complex models by teams which are multidisciplinary and geographically dispersed. The traditional solution is for participants to exchange information over the network (in long time scales) or by physically gathering around the model, which runs at a single powerful server.

This thesis defines *collaborative visualization* as the ability for multiple participants in a team to create, preprocess, run, steer, postprocess, visualize, and analyze a model and its' associated data sets interactively and collaboratively. It states a set of criteria for a good *collaborative solution* to this problem. These criteria are: a human time scale to all operations, collaborative, networked interaction with the model, and high reuseability of existing code. Existing work has produced solutions to meet all criteria, but not all at once.

A computation is viewed as a tree of values, where values encompass data, operations on data, and interfaces to data. A forest of such trees, with some leaves being facets of a single consistent value, can succinctly express both collaborative visualization and distributed computation. Key aspects of facet behaviour include consistency with peer facets, and transparent replacement of existing objects in legacy code.

Two prototype implementations of these *facet trees* are presented. The systems replace graphical objects in Matlab with glyphs. Glyphs exhibit facet behaviour, and are useful for constructing collaborative interfaces using existing Matlab code. The implementations are evaluated against an existing solution, T.128 application sharing, on the given criteria. Sustained performance on both sides was uneven. The sharing *between* instances of Matlab, however, seems to meet the collaborative criteria better than turn-taking over a single instance.

Contents

List of Figures

Acknowledgements

I thank all the people who have given of their time, energy and advice. In particular, there are my supervisors, Dr. Junwei Lu and Professor Chengzheng Sun, who have given support and encouragement, and caught my mistakes without stint. Special thanks go to the denizens of the Queensland Parallel Supercomputing Foundation (for the place to work and the supercomputer to play with), Andrew Lewis (for his sense of humour and bottomless well of obscure lore), and my family (who made it all possible).

Originality

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the dissertation contains no material previously published or written by another person except where due reference is made in the dissertation itself.

Aguido Horatio Davis

Chapter 1

Introduction

1.1 Collaborative Visualization

Pawletta [21] identifies three characteristics of contemporary simulation methodology. First, the models are computationally expensive to run. Second, the models are qualitatively more complex. Third, all phases of the modelling process require freedom of interaction with the work. Traditional programming languages cannot support this interaction as well as the *mathematical languages* such as Maple, Octave, Mathematica and Matlab.

These are built around the idea of interpreted steering, where user code in an interpreted language calls routines written and optimized in a compiled language such as C or Fortran. The decisions are made in code the users can rapidly develop and interact with, and the rest is supplied by the modelling tool. Mathematical languages are optimized for the manipulation of numbers and functions which operate on those numbers, which reside in a local workspace[1]. Mathematical languages work well for heavy numerical computation, so extensions to their computational throughput are useful, but none of those listed support concurrency in computation beyond threading on a single machine. To remedy this, there has been much work on equipping these languages with primitives

[1]For the languages listed, this is just a scope of execution. For some (Maple, Mathematica) a workspace is a document.

for parallel [20] [32] [30] and distributed [21] processing. However, the emphasis in this work has been the distribution of computational load across processors, not the distribution of computations among users.

Wood *et al.*[31] identify another key characteristic of contemporary simulation. This is that models are built, edited, used, debugged and analyzed by teams. They argue that this is a consequence of problems requiring several sets of expertise for insight. As these teams may be scattered geographically, personal collaboration over a single model site becomes infeasible and alternatives must be found. Agrawal *et al.* [23] discuss a large-scale multidisciplinary modelling effort, using distributed, heterogeneous human and computing resources. They then present the means they used to bind these resources together into a coherent whole.

These researchers have all attempted aspects of what we will call the *collaborative visualization* problem. Collaborative visualization starts by seeking to enable several users to steer and visualize the results of a simulation as it runs. Any solution to this problem must provide the infrastructure for a useful feedback loop between the users and the model.

One way to establish this loop is to run the simulation as a batch job, disseminate the numerical results as files to the members of the team, wait for their analysis and feedback, and then apply that feedback to the next run of the model, in however many hours it takes. For problems with expensive runs, where deep analysis and team consensus before committing scarce machine resources is a necessity, this approach is close to optimal. In times past this was the case for most computational science and engineering (CSE), and hence we will refer to this mode of work as the *traditional solution*.

With the increase in available computational resources, economy is not such an issue. Instead, attention has turned to the process of attaining understanding using the model and the data it generates. Experience indicates [21] [31][5] [1] that engineering and scientific problems are more easily understood if the practitioners can play with the problem. Environments supporting ad hoc anal-

ysis, visualization, and steering of the computation will therefore be more useful than the traditional solution. This steering requires a feedback time comparable to the informed reaction time of a human being, which in turn imposes requirements in modelling and visualization performance.

This *human time scale* is not necessarily real time. Some phenomena of interest, such as protein folding, happen over timescales of seconds. Others such as magma flow [12] and biogeochemical cycles [6] work over timescales of millennia. The human time scale must present these phenomena in a time roughly corresponding to the ability of the human user to appreciate them.

After each set of simulations, postprocessing, visualization and analysis of results and amendment of parameters or code in the model are in order. Postprocessing and visualization are supporting tasks to analysis, which benefits from multiple interacting perspectives. This interaction suffers considerable time and logistical penalties under the traditional solution. Editing needs to be steered by the consensus of the team, but must in the end be performed on one instance of the model. The traditional solution imposes a single editor. An environment which supports collaborative work in these areas, between scattered participants, will in turn foster insight and effectiveness in problem solving.

Accordingly, the ideal characteristics of the *distributed solution* to the collaborative visualization problem are:

1. Steering feedback is obtained from the users by the system, and applied interactively to the current computation, on a human time scale. This enables tuning of the simulation towards areas of especial interest without knowing where those areas are before the runs. It also allows the users to steer around problematic or uninteresting areas. To achieve the same ends without intervention in mid-run requires wasteful repetition.

2. The simulation runs on a human time scale. Simulations where the time evolution is too fast for meaningful human feedback vitiate the entire point of interactive visualization. Simulations where the time step is on

the order of hours or days are better suited for the traditional solution.

3. Information is extracted, postprocessed, disseminated and presented to the users on a human time-scale. This is a corollary of criterion 1.

4. Operations occur over a networked cluster of computers. In a realistic research environment the cluster will be heterogeneous and the network latencies large and variable. This implies several qualities in the solution implementation. It needs to be available on several commonly-used platforms, and able to be ported to new platforms as they arise. It must be interoperable between platforms. Local response should not be critically dependent on network latency. Distributed algorithms should be carefully chosen for reasonable performance under adverse network conditions, and should not assume the transport layer is reliable[2].

5. Authoring, model steering, and analysis tasks are collaborative. That is, multiple members of the team may interact with the same instances of things, and with each other, at the same time. Other characteristics are then implied, depending on the task, which may be found in existing work. A small selection are: consistency, intention preservation and causality preservation [28], group awareness and relaxed WYSIWIS[3] [2], unanticipated sharing and independent local work [31].

6. Existing modelling code is reusable. A primary advantage of mathematical languages is the legacy codebase. Quite a lot of this code is written by working researchers and engineers, and is thus hammered together from the raw mathematics using an undocumented evolutionary life cycle. Hence, any tools for parallel computation and visualization that interact with them must be immune to poor software engineering practice. Requiring no change at all in other code gives such immunity. Less ideally, minimal or rote changes in code give a degree of immunity.

[2]A convenient method of getting this quality is to build on top of an existing software layer, such as the Message Passing Interface.

[3]What You See Is What I See. See section 1.2.1.

These criteria point to a solution which preserves the strengths of the traditional solution but supports research in distributed groups. The rest of this chapter will survey various aspects of this solution, classifying them by design pattern and effectiveness. Chapter 2 will present the concept of *facet trees*, which provide a general framework for addressing criteria 1, and 4 through 6. Chapter 3 will discuss in detail some prototype implementations of these trees as additions to the Matlab language. Chapter 4 will evaluate the performance and useability of these implementations, both alone and in comparison with T.128-based application sharing technologies. Chapter 5 will present some conclusions, and identify issues for further research in this area.

1.2 Patterns of Collaboration

Distribution of a tool may be considered in two, roughly orthogonal axes: the interface and the computation. The computation may be distributed (MIMD[4], SPMD[5] or cluster computing) or centralized (a single application instance on a high-powered server). The interface may be distributed (each instance of the application interacts with several users) or centralized (each instance is owned by one user which it interacts with). Distribution on either axis may be transparent or explicit. Transparent distribution uses the existing code, and modifies the virtual machine in which the code executes for concurrency. Explicit distribution is accomplished by rewriting the code with awareness that the system architecture is concurrent and distributed.

1.2.1 Distributed Interfaces

Collaboration Transparency The International Telecommunication Union has promulgated recommendation T.128 [10]. This specifies a standard for multipoint application sharing, supported by the other recommendations in the

[4]Multiple Instruction Multiple Data distributed computing
[5]Single Program, Multiple Data, a subset of MIMD.

T.120 series. Application sharing[6] is the sharing of an application that does not know it is being shared. Each user sees a copy of the application interface, and generates input events (key strokes, mouse actions) as usual. A single instance of the application runs on a server, and is fed a serialized, merged stream of all the input events, which it processes as though a single user were generating them. In our classification, this approach is a *distributed* interface and a *centralized* computation.

Here, *transparent* collaboration means that a shared application instance sees the same environment, resources, and interfaces as an unshared instance. Legacy applications can then be reused with no change at all. With this advantage come a few weaknesses. If the application sharing system cannot rely on exposed information about application semantics, it cannot assume that certain actions may be safely omitted, summarized, reordered, or adjusted. It cannot then economize by transmitting only the ones the application needs to know about. It also cannot mix inputs from different users together - for example, the mouse events that make up a drag and drop operation must not be interleaved with those of another drag-and-drop. A user must therefore be designated to "hold the floor", working with the application while the users who do not hold the floor observe. The need to acquire the floor limits the kinds of collaboration possible.

Two popular application sharing systems are SunForum and NetMeeting. These are freeware built to the T.128 standard by Sun Microsystems and Microsoft, respectively. Both work as described. The users see a pixel-level copy of the application window, scavenged from the screen of the system which hosts the application. This implementation ensures that each user sees exactly the same view of the application - a property known as What You See Is What I See (WYSIWIS). The systems occupy exclusively such resources as the cursor, which makes independent local work infeasible. On the gripping hand, they may be used with any application with which they can coexist, and this makes them

[6] Or collaborative transparency as it is called in the literature.

a viable alternative for implementing collaborative visualization using existing mathematical language applications.

Mathematical Clients The client-server model can be specialized for the case of mathematical languages. A client application sends requests to a computational server, which does the heavy numerical lifting and returns results. Typically the client offloads rendering and user interaction from the server, which allows the two layers of software to be optimized for their specific tasks.

In the mathematical language Mathematica, a proprietary product of Wolfram Research, the engines are called "kernels". Each kernel runs on a single processor, and may respond to only one client. A client also lacks the ability to forward requests to multiple servers. Recently, a Parallel Computing Toolkit has become available. This contains bridging primitives for using a pool of running Mathematica kernels, but not for coordinating interfaces. A collaborative visualization solution built in this language can meet criteria 4, 2, and 1, but would entail rewriting existing code to be aware of multiple users. This *collaboration awareness* is something criteria 5 and 6 try to avoid in user code.

The TechTalk system attempts a similar approach, but built on the Matlab mathematical language. Matlab is implemented as a single process, so the project has added a Java client which runs within a Web browser and interacts with an instance of Matlab running on a remote server. All the clients in a single session use the same Matlab instance. The clients are collaboration-aware, which lets a set of students see each others' commands and graphical results. The system was originally designed for teaching, where students could work together on assignments remotely or lecturers could give demonstrations. There is provision for "detaching" a session, which copies the state to a Matlab session running on the client. A solution built on this technology could reach all of the criteria except 2, human scale run times. This is because of the single Matlab process doing the computation.

Visualization Clients Another specialization of the client-server model uses a separate, collaboration-aware application to present data generated by other, non-collaborative applications. These applications can include mathematical languages, and so a complete solution to the collaborative visualization problem can be built up.

SHASTRA[1] is a toolkit for building collaborative visualization applications over the Web. Distributed simulations are conceptually organized into DSAV (Data, Simulator, Analysis, Visualization) loops. Each software component of these loops can reside anywhere, linked by streams of data and events. The SHASTRA substrate, as the infrastructure is called, permits interactive, collaborative steering and visualization of SHASTRA-aware simulations. It uses a distributed set of servers linked by wide or local area networks. Support for floor and access control, group awareness and session management are integral. This toolkit has been used to build solutions that meet all the criteria of an ideal distributed solution to the collaborative visualization problem, except one. Existing visualization and simulation code needs substantial modification before reuse as a SHASTRA server (criterion 6).

CSpray [19] is a data visualization layer that draws visualization objects and information from existing sources, which require no modification. It is built from the Spray visualization tool, which implements *spray can* visualization. Spray can visualization uses that metaphor to visualize and transform a workspace of visualization objects generated from a data source. This lets users perform analysis and highlight interesting features. The system is extended for collaboration by equipping each user with their own can(s), and enabling users to share workspaces in an ad hoc manner. Group awareness, floor and access control, and local independent work are supported. Unlike SHASTRA, it can draw data from sources which are not aware of collaboration, but it can't steer them.

1.2.2 Centralized Interfaces

Traditional collaboration The traditional solution uses a centralized computation and a centralized interface. Visualization takes place on a specialized front-end machine. If the modelling team wish to collaborate, they must gather around the visualization machine and confer.

The computation may be distributed over a closely-bound cluster, or in a single parallel processor pool. Where there is only one process interacting with the user group, this is a centralized interface to a distributed computation. PV3[9] is such a system. Parallel data structures and nodes are instrumented to extract visualization data. The data converge on a visualization server. Instrumentation is done by modifying the simulation code to incorporate calls to PV3 libraries. This kind of technology lets solutions meet criteria 2, 1,3.

Cooperating instances It is possible to equip instances of existing applications with the ability to cooperate with other running instances of the same application. This establishes an explicitly distributed computation, as each user knows they own a separate instance. Each instance provides their own centralized interface to their user. In most cases the implementation uses the application semantics to economize on resource use.

Wood *et al.* [31] have applied this approach to the visualization of data sets. They build their collaborative visualization sessions from instances of IRIS Explorer, a conventional data visualization package[7]. They set several goals that such a session must meet. It must allow users to join and depart a session without prior notice. It must allow exchange of data between users at different levels of granularity, so that users only share what they want or need to. It must permit users to perform local tasks independently of the collaborative work, and to dynamically share or localize tasks as they deem appropriate. They meet these goals by adding data sharing and steering modules to IRIS Explorer. These modules are explicitly emplaced by the user in the visualization network,

[7]IRIS Explorer streams data from sources (such as files or running models) through a user-defined network of modules to preprocess, analyze and render it.

forming new data sources and sinks which extend the network to other places. The shared resources coexist with local resources and tasks. Data may be shared and unshared by rearranging a user's local network.

Mathematical languages may be equipped with the additional primitives necessary for parallel or distributed computation. The majority of this work has been done with Matlab, the most extensible of the extant mathematical languages, but other languages have some too. This approach lets a system meet criteria 4, 2, and 6 straightforwardly. It also supplies the building blocks for collaborative applications, but there are no projects which are tackling this yet.

The Cornell MultiTask Toolbox [32] builds a set of Matlab instances running on a processor pool into a coherent whole. There is one master and several slaves. The master is a single front end instance of Matlab, which presents the familiar console interface to the user. Execution of a parallel program is controlled by the master, and performed (mostly) by the slaves. The slaves communicate with the master and with each other through methods defined from and implemented in the Message Passing Interface (MPI)[16]. This interface is an open standard implemented on most parallel machines (which helps meet criterion 4). As there is no set of formal bindings for Matlab, the toolbox is a set of wrapper functions around MPI calls in C. Other wrapper libraries of this kind are MultiMatlab [30], the Parallel Toolbox [20], MPITB [11] and the Parallel Computation Toolkit for Mathematica.

The Matlab*P project [22] has produced a toolbox which uses Matlab as a front end to parallel code. The actual numbers in the Matlab workspace are replaced by opaque objects, which hold references to matrices distributed across a pool of parallel processors. These processors run dedicated parallel code, not Matlab. The parallel code is hand-written C and Fortran, some of which was generated by the project, and the rest as wrappers around standard parallel libraries such as ScaLAPACK. Definitions scripted in Matlab are supplied for the parallel matrices, which then integrate seamlessly into the language. The Matlab

front end and the slaves communicate using the Message Passing Interface. The net effect is a normal Matlab session with superior performance at computing with large matrices fast.

Matpar[24] is a similar toolbox which uses the Parallel Virtual Machine (PVM)[7] as a communications substrate to join a front end Matlab to a back end parallel computer. Standard parallel libraries like ScaLAPACK and the BLAS are used for numerically intensive linear algebra routines. These routines have to be explicitly called by the user, as they are not integrated as alternate definitions of the standard Matlab routines. It supports several parallel architectures, but only one at a time. However, because PVM supports dynamic process starting and stopping, this one back end can be a Beowulf cluster of lots of smaller machines.

Pawletta *et al.* have proposed a master/slave framework for gluing together simulation tools for distributed work[21]. This framework has been implemented for Matlab and several other modelling applications, which they have then used to build heterogeneous models. The substrate is the Message Passing Interface, and the component modelling tools may be distributed across the network. The implementation precludes the possibility of these tools presenting independent interfaces to different users, so it would only be useful as a back end to a collaborative visualization solution.

Cooperative editing Collaborative editing systems such as REDUCE by Sun *et al.* [28] and GROVE by Ellis and Gibbs [4] use an explicitly distributed computation[8] and a centralized interface at each instance. They aim to produce a single coherent document as the result, and thus impose a more stringent consistency model for the application state. Under the REDUCE model, the scattered instances of a shared document must eventually be identical (*convergence*), operations which depend upon the results of previous operations must happen in an order that respects this (*causality*) and operations must have the

[8]Distributed state, with operations on it, albeit with humans doing most of the operating.

net effect the originating users intended (*intention preservation*). Prototypes exist and are in practical use[28].

Collaborative editing is needful for a solution to meet criterion 5 as regards model amendment, and incorporation of such an editor as a component of a solution is straightforward. A less straightforward but perhaps more fruitful approach is to regard the mathematics and data sets behind the model as a document, and apply these existing technologies to interaction with them (see section 5.2).

Flexible collaboration transparency Flexible collaboration transparency is a variant of collaboration transparency proposed by Begole *et al.* [2]. They seek to improve the group aspects of collaboration by replicating the application on each user's machine and coordinating them externally. Each instance of the application sees the same merged input stream, and (hopefully) holds the same internal state. The interface is local, not broadcast, which improves network usage considerably. Under our classification, this is a centralized interface with a distributed computation.

As opportunities arise, components of the user interface are transparently replaced by variants which exhibit cooperative behaviour. In scrolling window panes, this cooperative behaviour enables users to have different views of the same application state concurrently. This *relaxed* WYSIWIS is a desirable feature for group use. The panes also give users an indication of where and what the other users are up to, visually. This *group awareness* is necessary for true collaboration. A cooperative document model enables users to edit the text contained in it concurrently, rather than waiting for the floor. This *concurrent work* is another valuable trait for group use.

Their prototype system, Flexible JAMM, manages this replacement in Java applets whenever they are serialized. The replacements are subclasses of the original Swing components which exhibit the required behaviour. Unreplaced components are handled as in conventional application transparency. Alto-

gether, the weaknesses in collaboration transparency are repaired while retaining the main strength, immediate legacy reusability, at the cost of restricting the domain of applications it can share. If this approach could be implemented for mathematical languages, it could meet criteria 3, 4, 5, and 6.

1.3 Scope of this Thesis

This thesis investigates the following approach to collaborative visualization:

- Viewing ongoing computations in mathematical languages as trees of dynamically changing values,
- identifying values that may usefully be shared with other trees, and
- introducing such infrastructure as will enforce the guarantee that these values will remain consistent.

The end result is a forest of what I call facet trees. The term will be explained in chapter 2. The questions to be answered are:

- Are facet trees a useful abstraction?
- Is it feasible to generally map existing modelling code to a facet tree?
- Is it feasible to do so in an application-transparent way?

An implementation of facet trees on top of the mathematical language Matlab was designed and constructed, focussing on visualization objects (graphics and user interfaces). This thesis covers the design and implementation of this prototype, and the ideas generated to explain and solve the challenges of this implementation.

Chapter 2

Theory: Facet Trees

2.1 An Example

Consider a simulation code which models the equilibrium flow of an ideal fluid. The simulation will run in a virtual machine (perhaps a Matlab session, or a Unix process, or a Java runtime). Inside the main simulation are some global variables (to store the geometry and boundary conditions), and several procedures to operate on them, and some interface objects. The heart of the simulation is a function from the geometry and boundary conditions to the velocity field of the fluid. Each procedure will have a name and be implemented as operations on a set of variables, and calls to child procedures.

To introduce collaborative visualization into this example, we must arrange for the interface to appear before several users, and the computations to occur in several places. Let us clarify a *place* to mean a thread of execution, the resources it runs on and the environment it runs in. We may then refer to a collection of places running a distributed computation as an *ensemble*.

This computation will require that certain values be visible to the users; they will want to amend the initial and boundary conditions, and view the stream solution. For input, they may edit expressions in text fields. These fields will evaluate to the values wanted by the function, and must be consistent; it would

not do for two collaborators to set different values at the same time, or to have the feedback appear before the amendment. The fields are likely part of a text box, which must contain the same fields at all instances of the interface. Hence the fields are the children of a larger object corresponding to the box. Wherever this parent is required, so must its children be, and their children (if any) and so on.

As with the input, so with the output. A simple way to visualize fluid flow is with a contour plot. This plot has the value of the stream solution, which must be consistent across the ensemble. Updates to this plot should respect causality, if the model is to be steered usefully. The plot will exist in a set of axes as a reference point. The axes will be sitting in a window. The plot will have siblings: a colour bar bearing the value of the mapping and a quiver plot of the gradient.

The plot is free to appear differently to different users. Attached to its primary value might be the knowledge that the range is -0.3 to 2.38, the contours are 15 units apart, the element size is three pixels by two, corresponding to five square millimetres in real life, and so forth. This is metadata - data about the data.

Points of similarity A class of things in the preceding example can be identified. Each of these things has a value. The value of a variable is the data it names. The value of a procedure is the function, or relation, between the inputs and the outputs of that procedure. One evaluates this value by supplying inputs. The value of a user interface is the data it is presenting or obtaining from the user. It can be thought of as a variable which is visible to the user. Along with the primary value, each thing also has some metadata, a mapping between names ("scale") and simple data ("1:28").

Each of these things has a parent and children. The parent of a procedure is the scope in which it exists - typically another procedure. The children of a procedure are the variables and procedures that exist within it. The children

of a variable are subranges of their data, of sufficient interest to warrant names and metadata of their own. As outlined above, the children of user interface components are those that exist within them. The whole forms a tree. For now, consider the root of the tree to be the virtual machine of that place.

Each of these things also exists in more than one place. That the interface objects exist in more than one place is trivial. If there is domain decomposition, code must be replicated at each place where there is data to be worked on. If there is functional decomposition, variables must be kept consistent from place to place as different pieces of code work on their values.

2.2 Facets

We have now characterized a place as having resources, a thread of execution, and a tree of mathematical objects: variables representing numbers, functions which act on those numbers, and graphical components which present those numbers. We term an object which exists in more than one place a *mirror*. By construction a mirror has at least one point of presence, called a *facet*. If this facet transparently replaces an ordinary object at the place where it exists, the legacy code that uses that object needs no change. This is their purpose. Two concrete kinds of facets have been investigated, taking common roles for objects in mathematical programming.

Runes A *rune* is a facet whose mirror encapsulates a value, and some information about that value. The primary value of a rune may be:

- A data structure of varying nature, which is used in computations.
- A computation, with or without side effects, following the functional programming concept of functions as first-class values.
- A value visible to the user, possibly susceptible to amendment.

The metadata, information about the primary value, is organized a set of keys and values. The minimum behaviours implemented by a rune are to evaluate and

amend the primary value, to add and remove metadata keys, and to evaluate and amend metadata values. If the primary value is viewed as another, mandatory metadata item with a predefined key, these simplify down to four.

Glyphs A *glyph* is a rune whose value is visible to a user. If the glyph's mirror has more than one facet, it is visible to more than one user. This makes them a building block for user interfaces. As with normal runes, a glyph's value is subject to update by user code, and by other facets. It may also be subject to amendment by the user, which allows user input to the simulation. This change in value may be designed to have the side effect of triggering an explicit response by simulation code, which allows user commands.

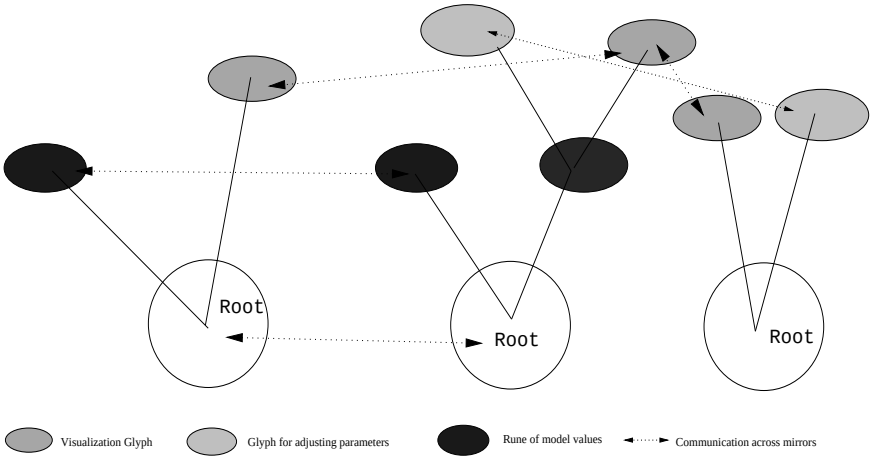

Figure 2.1: Three facet trees and three mirrors

Figure 2.1 illustrates the relationship between trees and mirrors, using the example of section 2.1. Three users each own a tree. The two on the left share a mirror which represents the running simulation. All three share a mirror which presents a contour plot; this mirror interacts with the computation via the middle tree. The model is steered by yet another mirror, shared by the two users on the right. It can be seen that the combination of tree (place) and mirror uniquely identifies a facet[1] It can also be seen that facets exhibit at least

[1] An interesting side issue is identifying the correct mirror for a newly generated facet. This

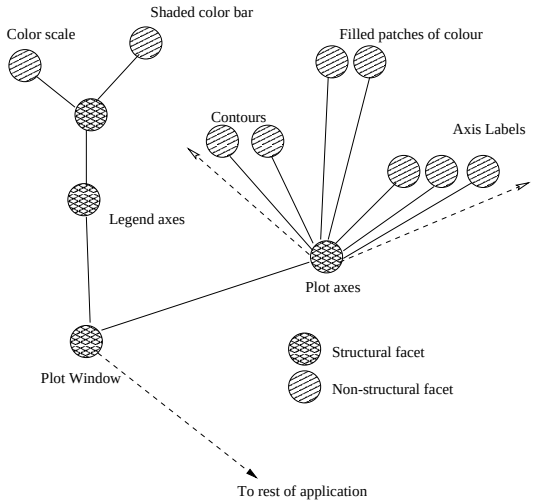

Figure 2.2: A plot as a branch

three separate sets of behaviour: towards other facets in their tree, towards other objects in their workspace, and towards peer facets in their mirror. The next sections will detail these roles.

2.3 Trees

The collection of facets at a place form a *facet tree*. A subtree, or branch, comprises a coherent subset of the shared objects. Places join the ensemble by creating the root of a new tree, which then then grows by grafting and budding. Facets within a tree are classified as structural or non-structural. This distinction determines whether changes in a subtree's structure are reflected to other trees in the forest. Structural facets define the local structure of a tree, and their arrangement should not be shared. Figure 2.2 shows a close-up of one branch of the fluid flow example of section2.1. The graphical window and the axes for the colour bar and contour plot are structural, as another user might

analysis of a model's values and their meanings would seem to require an agent with reasoning ability. For this work, we will introduce one: the author of the model. It is assumed that the human authors name values consistently ("Lava Temperature" here refers to the same "Lava Temperature" there), and the problem then reduces to finding and matching names dynamically.

omit one or have several. Having shared the axes with a contour plot in it, the facets corresponding to lines, filled regions, and labels are non-structural. When they are created or destroyed on one plot, that change must be shared with other instances of the plot automatically.

Budding is the replacement of an object in the local workspace with a facet, which becomes a child of an existing facet in the tree. If the bud is not a structural facet, each facet in the parent's mirror grows a facet of the bud as well, so that branches remain consistent in structure.

Grafting is the cloning of a branch from another tree. This clone, called a cutting, is attached to an existing facet of the local tree, and the mirrors involved then are extended by another facet each. This is how users can share things with each other - one creates the original, single-facet object, and others graft it on to their trees at an appropriate location. Two implementations of the cloning are possible.

1. Clone the base of the branch. Onto this base, graft each of the required children. In this way, some of the cutting is available immediately. Balancing this, the original branch needs to be locked against change during the transmission to prevent inconsistencies.

2. Create the cutting of the entire branch, transmit it to the graft site, and unpack it in one operation. This snapshot cannot be affected by changes at the origin, but takes more time to reconstruct at the new tree.

If backing instances are separate, the newly cloned facets must make arrangements to adopt or create them. Further discussion on this point is given in section 2.4.

Facet trees shrink a branch at a time. When a facet in the tree is deleted, any children cease to have a meaningful context and are deleted also. An example is a variable after the procedure that uses it has ceased - it has no purpose, and neither does anything *it* has as children, and so on. Another example is shown in figure 2.3. When the simulation code is told to stop showing a particular

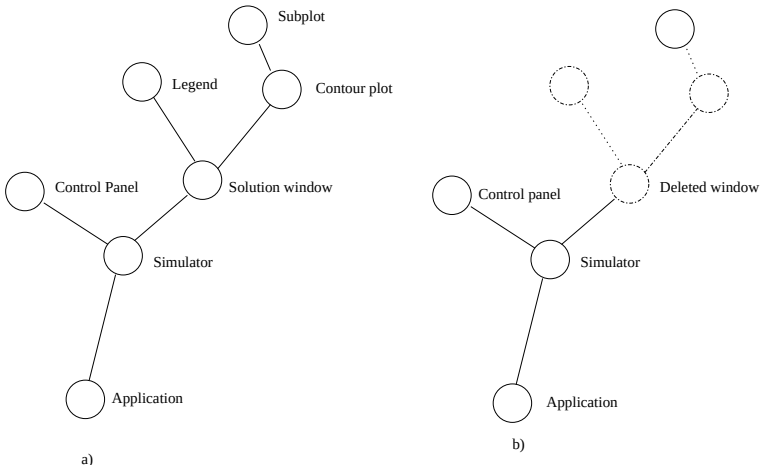

Figure 2.3: Removing a branch.

solution, it just closes that window. The rest of the tree remains intact and running. Deleting the root of the tree implies removing the entire tree from the forest, or removing a place from the ensemble. This is always initiated at the place (or user) which wishes to leave.

Runes Runes whose values are simple data tend to have few children (most ranges are not named) but several siblings (for example the H_x, H_y and H_z components of a magnetic field). Runes whose values are executable have backing instances of code, and these might have a deep hierarchy of function, sub-subfunction, sub-sub-subfunctions, and attendant variables. This profusion of instances will not be reflected in the facet tree, because most local code and variables will not be shared. Hence, branches of a facet tree made of runes tend to be shallow and flat.

Glyphs Glyphs are children of other glyphs, or of the runes which control them. Individual user interface components require a context of other components to convey information to the user (a lone text field with `42` in it could mean almost anything) and so they lead to fairly deep instance trees (3 or more levels). All levels of this hierarchy are subject to sharing, so these branches of

facet trees are also deep.

Roots The root of a facet tree has a number of unique attributes and roles. The backing instance for this root is the language runtime, because that is the parent of all the things generated by the user from code in that language, and it controls the screen, which is the parent of all the graphics. The runtime will have a set of global properties, and these will be reflected as metadata. It will have a primary value, which is the location of the place within the ensemble[2]. The root is therefore a structural glyph. The root is never a child of another facet.

The root mirror is named for the ensemble it underlies. To add a new place to the ensemble, a new root facet is created with that mirror name and merged in to the tree. The new peer is issued a location identifier for that place by one of the existing peers, which it uses to identify itself to the rest of the forest. These identifiers are small positive integers, which impose trivial overhead when labelling messages or elements of state vectors, and possess convenient properties and algorithms for generation and iteration. Several of the algorithms presented in section 3.3 use these properties to efficiently query or augment the set of peers in a mirror or trees in a facet forest.

The root of a facet tree provides a bootstrap naming service, which its descendants and peers use to acquire references to facets of other mirrors. Each local root knows the identifiers of its descendants. When asked for a reference to a facet of a specified mirror, the root fills the request with a local reference if it can. If it cannot, it delegates the request to the appropriate peer.

2.4 Backing Instances

A facet maintains exactly the same interfaces and visible state as the object it replaces. We will call a facet's implementation for this the *backing instance*. A facet may be its own backing instance. This is applicable where the original

[2]This value is immutable, and is held within the rune rather than the backing instance.

object may be superclassed to exhibit facet behaviour. If the exact interfaces are encapsulated where the facet implementor cannot get at them, or the original object may not be removed, or equivalent behaviour is impossible to reproduce, the original object must become the backing instance for the facet. Messages to and from the backing instance must be intercepted and modified by the facet. The backing application must expose enough information to permit this.

The set of objects in the local workspace forms a tree of potential backing instances, the *instance tree*. The same terminology may be applied to the instance tree as to the facet tree: parents and children, structural nodes, budding and grafting of instances. If the objects mostly form a flat collection, they can be all considered direct children of the root object, the application. A subset (possibly empty) of the instance tree will be replaced/augmented by members of the facet tree. Some of these replacements will be explicitly required by the user, some will be implicitly required by the user's code, and some will be implicitly required by the facet's code in order to create a coherent facet tree.

Explicit replacement occurs when the user creates the backing instance to be shared, indicating by tagging with a mirror name or by other means that it is to be augmented. If the parent of the instance is itself shared and there is no existing mirror as requested, growing a bud on the facet tree and supplying it with details of the binding at bud time is straightforward. If the parent is shared and there is an existing mirror by that name, then an entire branch must be grafted on. The backing instance of the root of the branch needs to be rebound to the newly imported facet, or replaced with a freshly created instance (which then needs grafting to the instance tree). Similarly, descendant facets within the branch must appropriate or replace backing instances of their own. A familiar example would be the creation of a two-dimensional contour plot, named and shared, such as the one in figure 2.4. This forms part of the branch illustrated in figure 2.2.

The entire hierarchy of window, axes, labels, colour patches, and lines would be created most efficiently in one hit, and *then* bound to the facet tree. At this

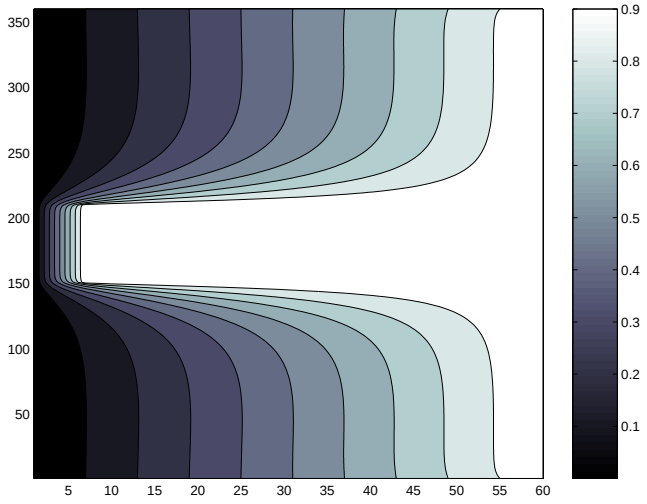

Figure 2.4: A contour plot

point, one of the named subplots might be discovered as a separate mirror elsewhere in the ensemble, and a complete branch arrive from it for grafting. If the backing instance for this branch consists of a blank, named pane of the window for it to occupy, the graft must be able to reconstruct backing instances for itself. If the subplot was created completely, from local data, the backing instances can be co-opted. This may or may not be more efficient than destroying the existing backing instances and reconstructing them from scratch. Additional issues exist where the interaction between the adopting thread (which runs the code inside the dormant grafted objects) and the backing application thread(s) is asynchronous. Enquiries to probe the instance tree may manage to deadlock.

Implicit replacement occurs when the user's code creates a nonstructural backing instance descended from a shared structural backing instance. If anonymous, this operation is always a bud, and the only complication comes from recursively budding new anonymous mirrors to deal with any prefabricated descendants in the instance tree.

Implicit binding to a freshly-created facet occurs when an explicitly shared backing instance has been created atop an existing but unshared instance. For the facet tree to cohere, the parent instance needs to be part of the tree. This

is accomplished by creating a facet to bind the parent instance, and recursing towards the instance root (the application) until a bound ancestor instance is reached. This traversal does eventually terminate provided the instance tree is actually a tree. Consideration needs to be given to the case where some of the newly bound instance(s) are structural. A consistent policy towards their unshared, nonstructural descendant is required. Sharing all these collateral branches may result in the entire instance tree being shared. Ignoring them might leave a necessary supporting value behind.

Runes The backing instance of a normal rune whose value is ordinary data, is a named variable in the scope of its parent rune. Binding consists of obtaining a reference and a copy of any properties.

The backing instance of a rune whose value is executable code is the code. Most mathematical languages are imperative and do not support direct first-class values, so running *instances* of the code are not explicitly named. The question then is, to what do you bind the rune?

The deletion of a rune from a tree might mean computation is no longer shared, so the corresponding backing instance is left intact, or it might be prompted by the destruction of that instance.

Glyphs Most mathematical languages possess a toolkit for general graphics, which supplies primitives like windows, axes, lines, menus, text labels, and more specialized things like patch objects and surfaces. These are the backing instances of a glyph. If the graphical components are presented in discrete windows, these windows are structural, so each user may have different things in different windows.

2.5 Mirrors

All the members of an ensemble share a common namespace, which requires a distributed, deterministic method for naming places and mirrors. All facets

in that ensemble with the same name are *peers* belonging to the same mirror. These peers communicate amongst themselves to maintain a consistent state and implement methods that are distributed in effect. Each peer may be doing something different, might have different local versions of the state, and need to be treated and identified individually. In contrast, the individual replicas of a replicated object are copies with no individual identity. They may be cached, relocated, or garbage collected as convenient for their distributed algorithms. Both kinds of object have state and behaviour, and algorithms in existing work to synchronize these are often exactly applicable.

A mirror starts with a single facet, which is a bud in a tree. Usually authors name their variables, procedures, and graphical interfaces. The corresponding structural facets which acquire these backing instances then have a meaningful mirror name. Non-structural instances are often generated on the fly, so not named explicitly. Such *anonymous* mirrors require a name which is unique over the ensemble for as long as possible, and acquire one by asking the root of their tree (see section 2.3).

However it is named, this mirror grows by merging with other mirrors of the same identifier. To do this, a subset of facets in each mirror acquire references to the facets of the other. In the present work, each facet holds references to the entire set of peers, and a check for existing mirrors is made as each mirror is created, so only an algorithm for the case of merging one facet into a set of facets is needed. Such an algorithm will be given in section 3.3. A mirror shrinks when one of the peers is pruned from its tree. This fact is broadcast to those peers it knows about (which in the present work is all of them). [3]

The two roles of a mirror are generalizations of the roles of an object: encapsulate some state, and operate on that state, possibly interacting with other objects to do so. Our two types of facets go about this in different ways.

[3]Maintaining a distributed set of peers by broadcasting updates to all of them, is an approach that may not scale well with increasing mirror size. For the current prototype, this is acceptable. Future work may limit each facet to a horizon.

Runes Runes make some of their state available to other local objects, notably their values. A mirror of runes whose values are data implements a distributed data structure. When this data structure is used in computations which must have correct inputs, and whose results should not be lost or dropped, the state should act as if reads and writes from all facets occur to one copy in a serialized order. This sequential consistency model [13] carries with it certain expenses. Some operations must be delayed until the operations ordered "before" them have been executed, or some of the state must be locked so that only one reader or writer is active. If the rune `stream` is passed as a parameter in a function call, `stream = gauss_seidel(stream, boundaries)`, then instead of considering the internal queries and updates posed to `stream` by the function `gauss_seidel`, we may consider this as one update[4]. At this time only relatively coarse functional decomposition of mathematical code seems feasible, so the locking overhead posed by such Lamport consistency will be negligible compared to the time spent computing with such "updates".

Runes operate upon their values (just as objects do when their methods are called). If those values are ordinary data, the operations may be implemented as a read followed by an update. If those values are meant to be evaluated by running them, some operations will involve execution of code in various places across the mirror. A mirror of runes whose values are actions then represents a MIMD parallel computation, each place supplying a separate set of resources, data, and possibly code. In this case, the backing instance will usually be an existing software component. The text of this code would be a metadata item, and any shared data structures referenced by it would be children of this rune. Synchronization would be done by blocking within these runes whilst distributed communication occurs. Evaluation of the value would either return the program text, or the result of running it, and such evaluations would have to be flagged as "execute if possible" or "evaluate only".

[4]In most mathematical code, the primary value is more likely to be updated and evaluated during computation than the metadata.

Glyphs Glyphs operate in a different environment than other runes. Their state forms inputs to processes, such as humans, which can trade transiently incorrect inputs against minimal update latency. Glyphs therefore allow for update of state to be local, and then to be brought into consistency as updates from elsewhere arrive. This optimistic concurrency model requires algorithms such as undo/transform-do/transform-redo[25], which roll back intervening updates, insert the operation, and roll forward the state again. For effective application of these algorithms, update operations must be fine-grained, reversible, and cacheable. For mathematical languages, updates tend to be total instead of incremental. Leaving no trace of the previous value rules out reversing them without a time-stamped history of values in storage, but that is expensive.

A callback is an item of metadata that is executable. When a certain event matching the callback key occurs[5], the callback value is executed. Glyphs may take user commands, and so need to support this additional operation on their metadata. Three cases may be distinguished.

- The callback is executed *classically*, with the exact behaviour of an unshared object. The facet executes a local version of the callback value. The other facets of the mirror are not notified of the callback event. This is to enable local, independent interaction of the user with their application, such as zooming in on or rotating a plot.

- The callback is executed *collaboratively* by all the facets in the mirror. The net effect is for the callback to execute *once*, at one facet across the mirror. Factors to be considered in choosing the place of execution may include: load balancing, migration latency, availability of requisite data and code. A good default heuristic is to choose the facet which most recently set the callback value, as being the likeliest to have a suitable context for execution. This is a distributed version of a classical callback, but exploiting the multiple places available.

[5]Typical events for a GUI are mouse clicks of several varieties, mouse drags and keystrokes.

- The callback executes *globally*, across all the facets in the mirror. The originating facet executes a callback intended to have a distributed effect, and notifies the rest of the mirror, which execute their own. The set of executing callbacks forms a MIMD computation. This requires equipping each facet with a callback value explicitly designed for concurrent execution. This callback value should be generated within the mirror from the application's supplied callback if the application code is to be kept unaware of distribution issues. How to do so generally is beyond the scope of this thesis.

Chapter 3

Design and Implementation: Concurrent Matlab

There are two ways to use facet trees as a framework for structuring collaborative visualization. If the majority of the functionality of the child runes rests within their backing instances, implementation can be simplified to a heavyweight root attached to the backing application, talking directly to backing instances inside the application. The rest of the framework then becomes a strictly conceptual aid to design, and is never instantiated. This approach was taken for the first prototype, detailed in section 3.2.

If the backing application supports an interface to a full programming language runtime, such as the Java Virtual Machine, and a lot of the functionality needs to be layered on top of the backing instances, then implementing facet trees literally, produces better results. No additional conceptual mapping from the system design to the logical architecture is needed. This approach was followed for the Java prototype, and is detailed in section 3.3.

3.1 Matlab as a Backing Application

Matlab is a mathematical language, a collateral descendant of FORTRAN. It has one type, a self-describing array of values. The values may be complex numbers, characters, structures with named fields, or other arrays. Variables are polymorphic. Object classes are created by naming a structure with specified fields and supplying definitions for the standard operators on instances of that structure.

The core interpreter, numerical, simulation and graphics libraries are proprietary to the authors, The Mathworks. The rest of the system is composed of interpreted ASCII script files, called M-files for their suffix. A collection of related M-files is called a toolbox. A core set of toolboxes is distributed with Matlab. There are many hundreds of commercial and freeware toolboxes written by working researchers and engineers in most fields of technical computing.

These toolboxes, which often define new types and methods as well as implement a large amount of applied mathematics, are seamless extensions of Matlab's capabilities. The built-in classes (double precision complex number, characters, and other primitives) may be redefined or overridden in the same way as any other. If compiled code is needed, there are standard interfaces for Fortran and C++, which let modules be used identically to M-files. The existing code base is one of Matlab's strengths.

The other strength of Matlab is the interactive environment. Here users may play with large and complex data sets, using the various toolboxes in an ad hoc manner to gain insight into the data and algorithms. This encourages extremely rapid prototyping (a standard method is to record an interactive session to a log, or "diary", and then edit the diary into the final source code).

Matlab's graphical capability is through opaque objects sequestered within the core runtime. Manipulation of these is through floating-point handles, so the system is referred to as Handle Graphics. Each class of Handle Graphics object has a set of properties, a mapping from keys to values. These properties

may be read-only or settable. Read-only properties give information on the object's implementation state (current cursor location, screen colour depth). The settable properties specify displayed data, field values, colour maps, positions, names, ages, and all the other things that determine what the object does and looks like. One queries these properties with the `get()` function, and updates them with the `set()` function. The system also updates them autonomously.

Each HG class has a specific constructor (e.g. `figure`, `line`, `patch`, `menu`). All classes share a common destructor, `delete`.

HG objects make good backing instances for glyphs, as metadata maps directly from the properties of the object, and a primary value can be identified for user interface components. Through Matlab's object system, enough information is exposed to use workspace variables as backing instances for runes. Most of the requisite communications primitives for facets are lacking, precluding an entirely native Matlab implementation, but as mentioned there are many programming interfaces for extensibility. Toolboxes for TCP/IP, socket I/O, and PVM operations exist.

With these traits, it was decided to use Matlab as a backing application for a facet tree implementation. Initially the compiled C API was used, linked with the Message Passing Interface. Presently the limitations of that approach became apparent (mostly single-threadedness) and Java was used.

3.2 Solution using MPI

3.2.1 Conceptual Origin

The resources available to the author were an IBM SP2 equipped with Matlab, and source for a pedagogical Matlab toolbox called ACVEM[14]. Ways were sought to use parallel and distributed computation in this toolbox. Initial work was in developing conventional parallel codes as solvers for the aspects of Maxwell's Equations used. In this work, it became apparent that the process had bottlenecks both in creating the model and solving the model. It also ap-

peared that the only way to ease these bottlenecks was to hand-code parallel and distributed extensions to Matlab in C and rearrange the Matlab code to allow for them.

A new approach was inspired by a technical report by Begole et al [2], which is reviewed in section1.2.2. This report describes the implementation of collaborative environments by leaving individual instances of the application unaltered, and introducing awareness (including collaboration-aware behaviour) into the user interface components and document models. This required less exposed information than would intuitively be needed. This is because the implementation language, Java, has strong support for interfaces. Objects can be subclassed and interfaces reimplemented to exhibit the required behaviour. These are then exchanged transparently without disrupting the running application.

It was then open to introduce the same kind of collaboration awareness into Matlab's user interface components, but by a different method. Handle Graphics is built into the core of the Matlab runtime, and the objects are not amenable to reimplementation. However, the entire relevant state of a HG object is exposed as a set of key/value pairs (the properties). Judicious interception of these values, by substituting for the commands which use them, would produce the requisite effects, if each instance was rendered by a separate Matlab engine. Unlike flexible collaboration transparency in Java applets, the work of computation would be done at one instance only, and the other instances sychronized by generated Matlab commands.

Concurrent Matlab over MPI is a toolbox that implements glyphs under Matlab. It provides the building blocks for collaborative interfaces to conventional computations: the computational back end is supplied by conventional Matlab code and compiled parallel executables, not runes. This toolbox is a set of C MEX-files for Matlab 5.2, running on an IBM SP2 and connected by daemons using the Message Passing Interface [16].

3.2.2 Architecture

Daemons The ensemble is a set of daemons written in C. A daemon is a program which lurks in the background and does not directly interact with users. The term arises from Unix, where system daemons provide services like print queues, network operations, and remote file systems. In this case, the daemons each run a Matlab Engine, an instance of Matlab whose standard input and output are under the control of the daemon. The engine interacts with the user and the daemon interacts with other daemons to coordinate the ensembles. These daemons are facets of a root mirror.

The basic reason behind this is the single-threadedness of the Matlab interpreter. It is not re-entrant. An attempt to feed it commands from another thread will wedge the application unless the interpreter is actually idle, a condition that requires ad-hoc methods to detect. There are auxilliary threads to parse events for graphical objects, but event handling involves the execution of callbacks, which are queued at the interpreter thread with all other computation. Hence, we will refer to the interpreter thread as *the* Matlab thread.

An alternative method of implementation is to use the MEX application programming interface to Matlab, which enables you to wrap a chunk of C code in a shared library which Matlab will then run as though it were a normal Matlab script. The incorporation of code using the Message Passing Interface to send and receive messages to another similarly equipped Matlab session works quite well. The problem is that this code is executed by the Matlab thread, and while it is engaged in blocking communications it is not engaged in servicing user events, degrading local response to an markedly level.

The POSIX thread library also works within MEX libraries, and appeared to offer a solution. A thread could be spawned from the Matlab thread, and listen for MPI communications, then use the existing `mexCallMATLAB` function to issue the appropriate commands as they arrive. This function's need to call the interpreter from another thread, and the issues involved using Matlab's thread-unsafe matrix primitives were prohibitive.

If asynchronous or ready-send semantics are used instead, the problem be-

comes one of remote response. As events arrive, Matlab commands may need to be issued in a timely fashion. These commands must be issued from code running in the Matlab thread, and hence this code must arrange to check for queued remote events on a regular basis. Either the Matlab thread must regularly enter the communications code at the behest of user code (which requires insertion of calls within user code) or it must stay within the communications code with occasional excursions to the interpreter. This latter involves substituting for the Matlab shell, a non-trivial task given its functionality, and feeding the resulting command stream to the interpreter. Local graphical events can be handled by regular calls to the Matlab command `drawnow`, which flushes event queues, but substantial queued callbacks may lead to starvation of the communications code.

The solution that was used is to run the daemon and the Matlab session as separate processes, and link them with Unix pipes using the supplied Engine API. This API implements a blocking master/slave relationship with an engine. The master blocks while the slave computes and sends a response back. In between times, the slave is free to respond to user input or other events. There is one daemon and engine per user (or at least per user interface), which may be running anywhere. The usual configuration is for each instance to be running on a separate processor of the SP2, the MPI communication to be routed through the high-performance switch, and the graphical windows to be routed through X11.

The glyphs are entries in a table in the root facet, which point to Handle Graphics objects. The root facet (the daemon) uses its backing instance (the Engine) to manipulate and render them as appropriate. The HG methods have been replaced with stub `MEX`-files placed in strategic places on the Matlab command path. They are called instead of the originals; they call the original methods, and then write an `mxRequest` containing sufficient information to duplicate the call, to a well-known pipe, where the `tap` thread of the local daemon will read it and it will be echoed to the rest of the ring.

3.2.3 Events

For simplicity in this first implementation, all facets were made non-structural, including the root. So each instance of the interface needs to exhibit the same appearance and behaviour. When an object is created at one instance, it needs to appear at the other instances. When an object is destroyed at one instance, it needs to disappear at the other instances. When an object changes properties, those properties must change at the other instances. When an event occurs to an object (mouse click, key stroke) it must occur to the other instances. When a piece of user code queries the properties of a HG object, it must get the same value from each instance.

These five requests - create, destroy, set, callback, and get - are the five method calls that a Concurrent Matlab node must exchange information with other nodes about. The minimum information to reconstruct a request is a Matlab array, a descriptive string, a handle designating the relevant object, and another object designating the relevant object's parent. These are implemented as fields of the `mxRequest` C data type, defined:

```
typedef struct {
   int kind;                    /* What kind of request this is */
   int status;                  /* Further detail on the above */
   char * words;                /* Associated name */
   mxArray * array;             /* Associated Matlab value */
   mxArray * object;            /* Associated Matlab handles (at root) */
   double parent;               /* Parent of object (at root)  */
   int origin;                  /* Which place originated this request */
   int location;                /* Which place the request is in now */
   int references;              /* Reference count */
} mxRequest;
```

This datatype is encapsulated in the `mpx` library, which implements the abstract data types for this system and most of the communication and reflection

logic.

Construction Glyphs are created with calls to the customized constructors, `Figure`, `Axes`, `Uicontrol`, `Uicontextmenu`, `Uimenu`, `Line`, `Patch`, `Surface`, `Light`, and `Image`. These are implemented as MEX-files linked with C threading, MPI and I/O libraries, and collected into a toolbox directory. Replacement requires an explicit user request. If the original constructors are called, the resulting HG objects will never be shared.

At the time, no method of redefining built-in primitives was apparent, so the replacements had sentence-case versions of the original constructor names. A similar approach is taken with regard to the other methods on a HG object, replacing with a custom version with a very close name and relying on user porting. This is not transparent, but does make porting code to Concurrent Matlab over MPI (in theory) a trivial search-and-replace operation.

The normal case is that a CM constructor is called with arguments that specify various properties of the resulting HG object. The arguments are passed unchanged to the original constructor, which parses them and produces the requisite object. Two of the constructors (`figure`, `axes`) may also be called to designate the current window or plot, and may do both if there are no existing plots or axes to so designate. The handle of the backing instance is extracted from the return values of the original constructor, and the handle of the backing instance's parent from properties of that instance. Changes to default properties for the instance are extracted by parsing the arguments to the constructor. The resulting list is packaged as the fields of a Matlab structure array. All this information is marshalled into a `mxRequest`. A well-known pipe (`/tmp/legion.pipe`) is then opened and the `mxRequest` written into it. At the other end of the pipe a `legion` daemon is listening. The `mxRequest` passes through some internal queues, as described in section 3.2.5, and is then broadcast to the other daemons as described in section 3.2.4. At each place, the appropriate thread of the daemon unmarshals the `mxRequest` into the constructor name, the parent

backing instance, and the properties it is to have.

At this point adjustments must be made for the local context. HG handles are generated in a nondeterministic way, which means that each facet's backing instance has a different handle. The name of a mirror is taken from the handle of the first facet to be created, which is always at place 0. Translation between mirror names and local handles needs to be made before these are used to designate the parent instance and any other metadata that points to objects. For this purpose each daemon keeps the state of the local facet and instance trees in a dictionary, defined as:

```
typedef struct _object {
   double handle;               /* What this instance of Matlab calls it */
   double rootID;               /* What the root calls it */
   double rootParent;           /* What the root calls its parent */
   char * callbacks[mxMAXCBACKS];    /* Collection of callbacks */
   int status;                  /* What are we doing to it? */
   struct _object * next;       /* Away from head of list */
   struct _object * previous;   /* Towards head of list */
} mxObject;


typedef struct {
   char * name;                 /* What it's called */
   mxObject * start;            /* Head of the list */
   int count;                   /* Population of the list */
   pthread_mutex_t * gate;      /* Gatekeeper mutex */
} mxDictionary;
```

The constructor is then called, and the handle of the resulting backing instance recorded in the dictionary. A set of metadata is created from the `mxRequest`, plus a set of callbacks which point to the system. This set is then loaded into the backing instance(s). The newly created objects are then entered

into the `mxDictionary`.

The second common case occurs when a Handle Graphics constructor is called to create a child of a backing instance that doesn't actually exist yet. For example, a call to create a line requires a set of axes for it to exist in, and will create one if a suitable parent does not exist. The axes will require a figure (window) to exist in, and ones will be created if suitable grandparents do not exist. The custom constructor stub which called the original cannot tell whether this implicit creation has occurred, but needs to ensure that the relevant facets exist. Therefore it always extracts the parent [1] from the properties of the new child instance, and then forwards creation `mxRequest`s for them. The daemon consults its dictionary and silently discards creation requests for existing facets. To avoid this issue at the other facets, the creation requests are sent in descending order - grandparent, then parent, then child.

Destruction Glyphs are destroyed by a call to the `Delete` command, which takes an array of handles to dispose of. As with constructors, the first thing it does is feed the input parameters to the corresponding local command, `delete`, which kills off the relevant backing instances. Since this command is only ever issued from user code, and user code only ever runs at location 0, the handles passed in are the mirror identifiers and require no adjustment. If the execution throws no exceptions, then an `mxRequest` with the handles in it is created and written to `/tmp/legion.pipe`.

At each node after broadcast, the mirror handles are translated into local handles, and the new list is passed to a local `delete` command, which disposes of the local backing instances. The corresponding entry in the `mxDictionary` is then removed.

Deletion follows the normal instance tree semantics, which means if a figure is deleted, any axes within that figure go as well. If a set of axes is deleted, so are any child graphics objects. Passing a deletion request for the root of

[1] And grandparent, if necessary.

each branch suffices, as Matlab implements this behaviour automagically. The dictionary at each tree needs to have the entries for the implicitly unbound facets purged; this is done by checking the parent of each entry.

The screen is considered to have handle 0.0. A request to delete it is a signal to terminate the ensemble, or magic bullet, usually fired by the user at location 0. As the bullet is broadcast to each facet of the root, it proceeds through all the threads in that facet. As the bullet passes through each thread, the thread must clean up any pending input, send the bullet on, then shut down. The last thread in the facet sends it to the next place on the MPI intracommunicator. The last facet to receive the magic bullet is the origin of it, and can then shut down itself.

Properties Most glyph metadata is held in Matlab as the Handle Graphics properties of the backing instances. It is queried by user code using the `get` command, or, under Concurrent Matlab, `Get`. As soon as a change is made in these properties, the change is made across the mirror. This eager update makes queries strictly local. Therefore, `Get` is implemented as a simple call to `get`.

Metadata is changed using `set`, or under Concurrent Matlab, `Set`. Normally `Set` takes as arguments an array of graphics handles and a list of keys and values as arguments. These arguments are fed to the normal `set` command. Assuming the local update is successful, the arguments are parsed into a handle and a Matlab structure array containing the key/value pairs. This array and handle are packed into an `mxRequest`, which is written to `/tmp/legion.pipe` for broadcast. As each facet receives the `mxRequest`, it extracts the handles and looks up the relevant facets in the object dictionary. Using the information in this dictionary, the backing instances are found and the supplied metadata values adjusted. A call to `set` is issued to update the backing instances.

Most metadata values for HG objects are character strings, numeric arrays, or handles pointing to other mirrors. For example, a set of axes travels with a

set of text labels corresponding to the tick marks, which are pointed to by the `XTickLabel` and `YTickLabel` properties of the axes. Before reflection at places other than location 0, any such handles need to be translated by looking up the local facets and finding their backing instances.

Callbacks A callback in Matlab is a command string designated for execution at the occurrence of a specific event to a specific HG object. In regular Matlab they are stored as properties of the object in question. When a user interface object (menus, buttons, sliderbars, text fields, and so on) has its primary value changed, Matlab classes this as an event and fires the `Callback` callback. When figures are resized, Matlab retrieves and executes the `ResizeFcn` callback. In all cases, all the other properties of the HG object in question are also updated.

In Concurrent Matlab, callbacks are stored in the `mxDictionary` of the root facet which originally created the object [2]. The relevant properties of the backing instances are occupied by a string containing the `Callback` command. This command takes parameters supplied when the event is triggered, specifying the facet which is involved and the callback key. The latter parameter is implemented as an integer from the set of constants defined in `mpx.h`, for simplicity of processing.

When a callback is fired, `Callback` takes the handle and key, extracts the primary value from the backing instance using the handle, puts these in an `mxRequest`, and writes it to `/tmp/legion.pipe` for broadcast. For group awareness, as each places sees the `mxRequest` it extracts the primary value and updates the relevant facet as for `Set`.

Callbacks execute collaboratively (see section 2.5). In this implementation, they execute at the location where the mirror was created. As currently configured, all mirrors are created at location 0, so that is where all the computational work triggered by the GUI occurs. When the root at that location sees an `mxRequest` bearing a callback it retrieves the real callback string from

[2] Most of the time this is location 0

that facet's entry in the dictionary, and issues it as a command to Matlab. For simplicity the callback archive is implemented as an array of string arrays.

3.2.4 Communications Topology

The designed function of the communications topology is distributed nonblocking broadcast. Each place needs to have the same communications load, each place needs to see all `mxRequest`s, and the thing needs to scale simply. Nonblocking broadcast is specified to avoid deadlock without the overhead of detection and avoidance algorithms, and to minimize synchronization requirements between daemons.

The Message Passing Interface [16] broadcast functions requires synchronization of the entire communicator, so it was decided to hand-code nonblocking broadcast using a ring. All daemons are members of a one-dimensional Cartesian intracommunicator, established at startup. For convenience one place (with rank 0 on the communicator) is elected as the root location, and presents a simple command prompt; commands entered here are executed by the local Engine. MPI does not allow communicator membership to change, which fixes the places in the ensemble at the start. This characteristic is a drawback which is addressed by the MPI 2.0 standard [17].

Figure 3.1 shows the communications topology. As `mxRequest`s are generated, they are passed from the outbound queue of one daemon to the inbound queue of its neighbour using nonblocking point-to-point MPI calls. The FIFO nature of these queues gives rise to two useful properties. A set of `mxRequest`s raised in some total order T at a given place pass through any other place in that same order. A set of `mxRequest`s raised at the same actual time at places arranged around the ring in some order O pass through any other place in the ring in a cyclic permutation of O. That is, if requests are raised in places 1358, one place may see them in the order 8135, another may see them in the order 3581. If `mxRequest`s are commutative, the results converge. If none of the requests conflict (same mirror, same metadata, different facets) then they also converge.

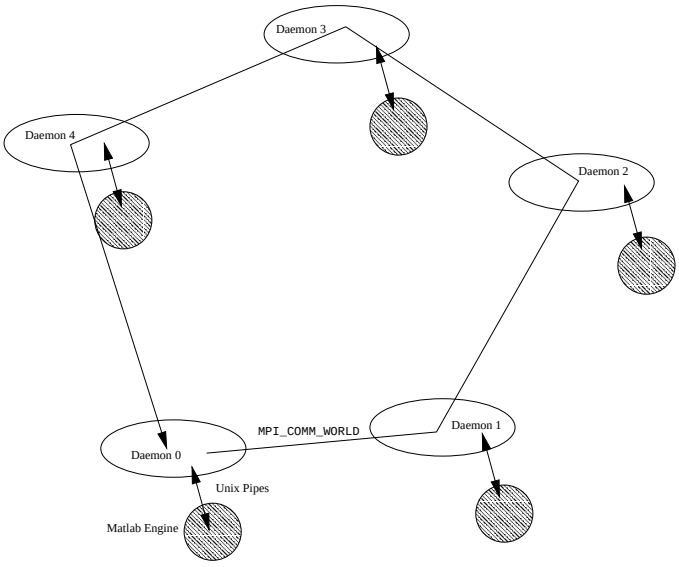

Figure 3.1: A ring of Concurrent Matlab nodes

In general, however, conflicting commands do not give rise to convergence (see section 4.2).

3.2.5 Threads in a node

Figure 3.2 shows the internal structure at a place. The daemon is split into four threads, named `keyboard`, `tap`, `engine` and `messenger`. Linking these four threads are two FIFO queues of `mxRequest`s, the `inbound` and `outbound`. The arrangement is pictured in figure 3.2.

Message passing thread The `messenger` thread deals with the MPI intracommunicator. It uses the LAM/MPI library [3] because this library is portable and supports heterogeneous clusters of hardware. On the down side, it is definitively not thread-safe. At process startup, this thread sets up the `MPI_COMM_WORLD` communicator for use, opens a Matlab Engine, and spawns the other threads. It then operates as a finite state machine, starting in state `WORKING` and looping through two tasks on each transition:

- Receive one `mxRequest` from the right neighbour in the ring and place it

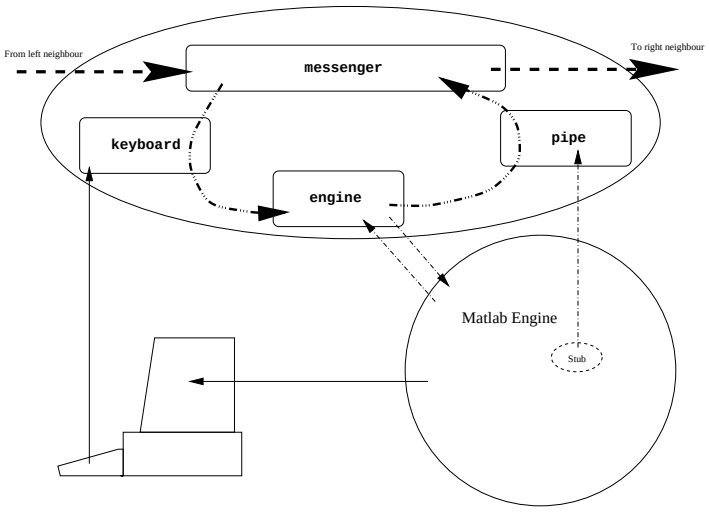

Figure 3.2: Implementation of a Concurrent Matlab node

on the inbound queue. Discard requests that originated here (they have circled the ring). If in `LISTENING` state and the request is a magic bullet, change to `STOPPED`. Otherwise, stay in the current state.

- Remove one `mxRequest` from the outbound queue and send it to the left neighbour in the ring. If the request is a magic bullet and it came from here, change to `LISTENING`. If the request is a magic bullet from elsewhere, change to `STOPPED`. Otherwise, stay in the current state.

The transition diagram is shown in figure 3.3.

Keyboard thread The `keyboard` thread is responsible for managing standard input. One user, at location 0, has a prompt presented to them. Lines from this prompt are parsed by a simple interpreter. The command "quit" fires a magic bullet, empty commands are ignored, and all other commands are passed to the backing Matlab instance verbatim. Commands from this source are wrapped in an `mxRequest`, and placed on the `inbound` queue.

Engine thread The `engine` thread is passed a pointer to an Matlab Engine when it is spawned, and is thereafter responsible for managing it. While there

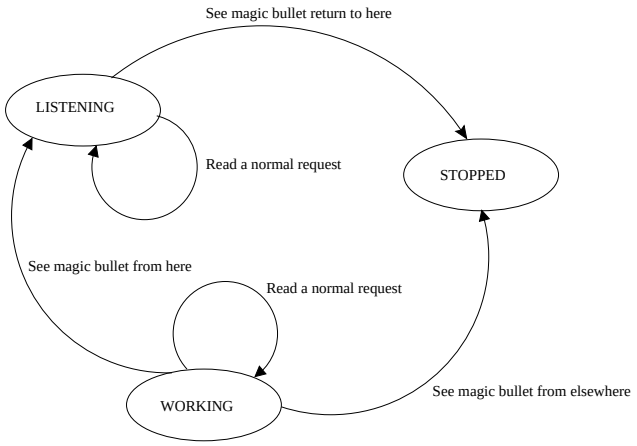

Figure 3.3: State transitions within the listening thread.

are `mxRequests` in the inbound queue, it removes one at a time and passes it to a library function which translates it into Engine commands and extracts some Matlab output. If this thread is running at location 0, and the `mxRequest` originated at standard input, the results from Matlab are printed to standard output and the request is silently destroyed. If this request is a magic bullet, this thread signals the other spawned threads to cease. Otherwise, the `mxRequest` is placed on the outbound queue for the `messenger` thread to continue broadcasting. No requests are ever reordered, only absorbed.

Tap thread The `tap` thread is so named because it sits on the end of a pipe. It is necessary because the Matlab Engine API only provides synchronous communication in one direction, from master process to slave Engine. The standard Unix pipe `/tmp/legion.pipe` provides the reverse channel. This thread is passed the file descriptor on creation. It then functions as a finite state machine. On each transition, it does two things

- Read an `mxRequest` from the pipe. Adjust the local dictionary as necessary (as it has just been executed by the stub that wrote it) and then put it on the outbound queue.
- Check to see if the `engine`) thread has signalled a state transition.

When it hits the `STOPPED` state, it terminates.

Note well, no ordering is enforced between the two sources of `mxRequest`s for the outbound queue. To do so, both `engine` and `tap` threads would have to see all the requests on the queue. There is no simple way to distinguish between a request meant for execution here and one passing through after already having been executed, which makes this infeasible.

3.3 Solution using RMI

The system described in this section (known as Concurrent Matlab over RMI) was created to test the feasibility of using facets as a design pattern. In this prototype, the framework is directly translated into the corresponding Java objects. The facets are Java objects, the glyphs are HG objects. Java threads run in the same virtual machine as a native thread which implements the Matlab kernel, gaining the advantages of concurrency without the performance penalties of communication with a separate daemon process. In particular, HG objects can be equipped with direct references to these Java objects in their metadata, which means a great deal of the system can be written in Matlab. The facets communicate with each other through Java's remote method invocation mechanism.

The system consists of two Java class hierarchies (shown in figure 3.4) and one set of Matlab class definitions. These will be described in turn.

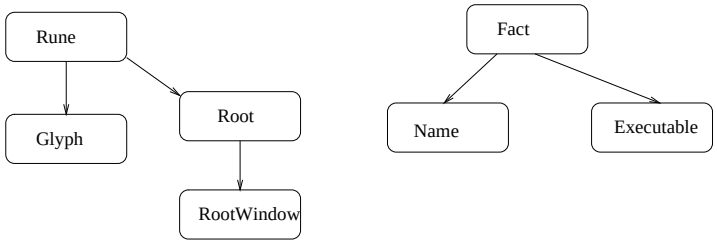

Figure 3.4: Class hierarchies

3.3.1 Metadata

Facts Metadata and primary values travel as instances of class Fact, which is a wrapper around a (key,value) tuple, and two optional attributes. These are a location (a small nonnegative integer) and a payload (a list of other facts). Locations of value `NO_LOCATION` are wildcard or unassigned. `Fact`s with no value are possible. These incomplete facts act as wildcards and query templates.

The normal case is for the facet to store a hash table of `Facts`, one per metadata item. Each `Fact` has a key, value, the location of the facet, and no payload. On metadata update, the relevant `Fact` is broadcast to the rest of the mirror. When a peer receives a fact, it discards or accepts it (and signals this to the originator), based on causality, value validity and key validity criteria. If accepted, it is stored as the new metadata value and reflected to any separate backing instance.

Certain facts form a coherent subset of the metadata and need to be updated in synchrony. The specific motivating example was the primary value of a Matlab surface object. Each vertex of a surface has three coordinates and a colour, specified by four separate properties. Changing one (especially, specifying a different number of points to the number specified by the other properties) is unwise. Updates from the backing instance are therefore processed conventionally, and then packed as the payload of a single `Fact`. Upon receipt, each `Fact` in the payload is processed as above and accumulated for simultaneous update of the backing instance.

There is provision for time stamping with an instance of class `StateVector`, which implements a standard state vector [13][28]. Only `Fact`s which have the right time stamp (are *causally ready*) for execution are processed immediately. The rest are queued internally. The queue is examined when a new `Fact` is presented and causally-ready `Fact`s are executed. In the present version, all `Fact`s are always causally ready. When a mechanism for tracking and updating the current state vector is created, this test will be made operable. The question to be settled is what scope a state vector has - one facet or one tree?

Names A `Name` is a `Fact` which binds a mirror or tree identifier to a reference to a Facet. Each facet keeps references to its peers as a hash tables of `Name`s. The `Name` of each facet in a tree is kept in a hash table by the root of that tree. Incomplete names (binding an identifier to a null reference) are equipped with the ability to complete themselves using the resources of the facet making the enquiry. The implementation of this *search by visit* runs as follows:

1. An `Ancestor` A is called upon to complete a `Name` N.
2. The facet calls N's `complete` method, passing as parameters a set of peers P and a set of `Name`s R which might contain the key of N.
3. If N is complete, it stops and returns itself.
4. If N is not complete, it attempts to find a complete `Name` N' with the same identifier from the list R.
5. If N' exists, it contains the value needed to complete N. N copies this value and the location from N', and returns itself.
6. If not, N switches state to `SEARCHING` and selects one or more peers of A from P. For each peer, it calls the `complete` method with N as parameter, and the search proceeds recursively.

Currently, the algorithm is implemented as a broadcast query. A `Name` in the `SEARCHING` state checks the peer it is sent to, and if it can't find a value, gives up instead of continuing on another peer. Entirely different searching behaviour can be constructed by creating a subclass of `Name`, equipped with different states and another selection algorithm (possibly traversing the n-ary tree which is labelled by the location identifiers, which would be more efficient).

Executables An `Executable` is a `Fact` whose value is code or a reference to code. That is, when the value is evaluated, it may have the side effect of starting a computation in the place where it is evaluated. For this purpose it is equipped with a slot for an alternate value, and an internal flag which defines whether

the `Fact` executes globally or locally. The value the `Executable` exhibits varies in accordance with this flag's value.

3.3.2 Facets

Facets are objects descended from class `Rune`. `Rune` stores metadata as a hashtable of `Facts`, the peer list as a hash table of `Names`, a reference to a parent in the tree, a reference to the ancestor of the tree, the location of the tree, and a (possibly empty) list of children. With this information it implements three interfaces, corresponding to the three sets of behaviour expected of a rune. These are detailed in the paragraphs below.

A `Rune` has three states, active, inactive, and dead. It starts off `INACTIVE`. When it successfully adopts a tree, mirror, and backing instance, it become `ACTIVE` and responds to requests from the outside world. When it no longer is part of a tree, it becomes `DEAD`, withdraws from the mirror and unbinds the backing instance, and awaits garbage collection.

Peer `Peer` is an interface containing the interactions between peer facets of a mirror. Peers are by definition in other places, so this interface extends `java.rmi.Remote`, to make these methods remotely invokeable.

Peers add or remove facets from their list of peers. The current scheme is a two stage broadcast using the `add` method. This method takes two `Names`, one belonging to the candidate facet which wishes to join the mirror, and one belonging to a *referee*. The broadcast proceeds as follows:

1. The candidate C uses the naming service provided by the registry to find a facet R of the mirror with the same identifier as C.

2. `add()` is called on R with candidate C and referee also C[3].

3. R acts as referee to C, calling `add()` on each of its peers, with candidate C and referee R. When the broadcast is finished (and the timing is

[3]In this discussion the `Name` of a facet and the facet are used interchangeably, where it is clear from context which is used.

important), C is added to R's peer set.

4. As each peer P receives the invocation of `add`, it adds C to its peer set, and then calls `add()` on C with P as candidate and no referee.

5. As C receives each invocation of `add`, it adds that P to its set of peers.

At the end of this process, two conditions hold. First, every facet in the mirror is aware of the new peer. Second, the new peer knows every one of the existing peers that is aware of it. It can be shown that these properties hold even when several candidates are joining the mirror simultaneously. As an optimization, the initial referee R prepares a `Fact` which summarizes the mirror state and passes it back as the result of the initial call to `add`, for rapid synchronization. When a peer is pruned, it calls `remove` on each facet of its mirror that it knows about (which are all the peers which know about it).

Peers evaluate and amend metadata belonging to the mirror. When a peer updates a metadata value, it is packaged in a `Fact`. It then calls the method `amend` on each of the other peers with this `Fact` as argument. If the peer is in the `ACTIVE` state and the `Fact` is complete, it is checked for causal readiness as outlined in section 3.3.1. Causally unready facts are added to a queue maintained by each facet[4]. Causally ready facts are added to the metadata table, and *then* reflected to the backing instance, and then success is returned to the originator.

A peer may request that another peer evaluate a metadata value it holds, by calling that peer's `evaluate` method with a metadata key as argument. The remote peer then retrieves and evaluates the value it associates with that key, and returns the results. A second argument supports an optional side effect; if the relevant fact is actually an Executable, this argument may specify a mode of execution, in stead of evaluation. This method was originally intended to be the dual to `amend`, enabling peers to enquire of each other their values. This is not really required with the present broadcast-on-update scheme, but the side

[4]All facts test as causally ready pending further work.

effect (indicating a peer's interest in that particular `Fact`) is used to implement distributed execution in glyphs.

A peer can supply a clone of itself for grafting, via the `cutting` method. This cutting will have the same mirror identifier, metadata, and primary value. The cutting is made via Java Object Serialization, which creates a byte stream that can be reconstructed portably into the object. References to other objects are handled by serializing them into the stream as well, and rebuilding the reference graph on deserialization. This stream is then wrapped in an instance of `MarshalledRune`, and returned as the result of this method. The wrapping would not be necessary, except that the children are objects that may be accessed remotely (`Rune` implements `Peer`) and by default the unpacked cutting is equipped with references to the originals rather than local copies. Unpacking them by hand ensures the references are to local children. Section 2.3 outlines another method, but finding a base case for the recursive grafting of an entire branch is problematic.

Child `Child` is an interface containing the operations supported by a child node in a facet tree. Children are created in `INACTIVE` state, either when unpacked as a cutting, or when budded. Such facets ignore (return a null result for) any operations on metadata, but do participate in update operations on their mirror population. A child transitions to the `ACTIVE` state when it is supplied with

- a `Parent` in the facet tree,
- an `Ancestor` at the base of the facet tree[5],
- the location of the facet tree, and
- information for binding a backing instance.

These things are passed in as arguments to the `adopt` method. This ensures that any facet in the `ACTIVE` state is equipped to interact with the tree, mirror

[5]Which is *always* the same as that of the parent.

and backing instance, removing the need for intermediate states where some of this information has been supplied and some hasn't. Firstly, the new facet uses the name services provided by the ancestor to seek a mirror. If one already exists in another place, the child uses the `add` operation in the `Peer` interface to join it and get a copy of the mirror state. If one already exists *on this tree*, the adoption is performed by the elder facet, and this one disposes of itself. If none exists, the child equips itself with some default state. Secondly, the child integrates itself into the tree by registering with the parent and ancestor[6]. Thirdly, the child takes the information about the binding instance (a generic array of `Objects`) and feeds it to an internal method. In subclasses with external backing instances, this method is overridden to do the binding. In `Rune`, it has a null implementation. Then the child changes to `ACTIVE` state.

If the child is already in `ACTIVE` state, the call to `adopt` is interpreted as a request to move from the old tree location to the new one. This is done by unbinding from the old parent and ancestor, and registering with the new ones. No mirror operations are necessary. The instance tree might need adjusting, which is delegated to another hook method with null implementation in `Rune`.

Active children, or inactive cuttings, may have descendants. These should be informed of the new ancestor/location/backing instances, and this is done by calling the `adopt` methods of each child. This recursively informs the entire branch. Normally these children do not already exist in the tree. If they do, then the call to `adopt` replaces them with the existing facets as for any other `ACTIVE` facet. If they exist, and are ancestors of the graft site, this creates a cycle in the tree. In this case, few of the properties of a facet tree still hold, including termination of the algorithms. Checking for this contingency would require a traversal from the graft site to the root, and a check of the results against the mirror identifiers of each descendant, which is expensive. A minimal check against adopting a facet as its own parent is in place.

[6]If this happens first, the ancestor will erroneously indicate an existing local facet of the mirror, the one which just registered.

Parent `Parent` is the dual of `Child`, containing the operations supported by a parent node in a facet tree towards its children. Facet trees grow by requesting a parent to add a facet of a named mirror to their children. This request is made by calling the `graft` method of the parent, with the mirror identifier as argument. The parent then consults the distributed name service (via its `Ancestor`), seeking a facet of that mirror.

If one is found on another tree, a cutting is requested from it and unpacked. The cutting then adopts the graft site as parent (see the `Child` interface). If a facet is found on this tree already, it is shifted to the new graft site. If no facet is found at all, a new mirror is budded with that identifier.

A parent takes updates to the list of children via the `adopt` and `disown` methods, which add and delete children from the list. Note that while growth of a tree is initiated from the graft site (a `Parent`), pruning of a tree is initiated from that which is to be pruned (a `Child`).

Ancestor `Ancestor` is an extension of `Peer` which adds the operations supported by a root node in a facet tree. Ancestors complete names and generate new identifiers. The base implementation of this interface is the class `Root`, which is a subclass of `Rune`. `Root` overrides the methods in child with null implementations, because the base of a tree is never budded or grafted.

Descendents of the root may create an incomplete `Name`, and pass it as argument to their ancestor's `complete` method. This method then undertakes to find a reference to the named mirror, and complete the `Name`, using the search-by-visit mechanism outlined in section 3.3.1. In `Root`, this is implemented as a call to the Name's `complete` method, with the root facet's peers and name list as argument.

The name list is compiled by the root, from the calls to the `bind` method made by children during adoption. This method takes a `Name` as argument. In `Root`, it simply adds it to a hash table of other such `Name`s. Facets which are changing to `DEAD` state call the `loose` method with their `Name` as argument, and

are removed from the list.

New identifiers for trees are location numbers. They are generated by treating the name space as a b-ary tree. The first tree in the forest has location 0, and descendants with locations $1 \ldots b$, where b is a constant. A tree with location n has descendants $nb + 1 \ldots nb + b$. A location may either be *occupied* (there is a tree there), *vacant* (there is definitely no tree there) or *requested* (the location has been given out, but no facet from there has joined the root mirror). Upon a request for a location, a root first seeks a vacant location among its descendants to satisfy the request. If it finds one, the location is marked *requested* and returned. If no descendant locations are vacant, an occupied one is chosen arbitrarily and the request delegated. If there are no descendants (the requisite location numbers are greater than a system-defined maximum) the request is delegated to location zero. If location zero is not occupied, it is the answer to the request.

This algorithm has been designed for a balance of robustness and speed. A location is found in at most $\lg(n)$ steps (the height of the location tree). A central name server could do the job in constant time (neglecting congestion) but represents a single point of failure. A root facet directly administers at most b locations, its descendants, and is subject to replacement by the other facets in the mirror if it goes silent. This property requires that each peer facet knows of *all* the other peer facets, making vacation of nodes trivial to track. A weakness is the generator of last resort, location zero. It is the origin of the ensemble, and will always be replaced as necessary, but two simultaneous overflows to it when the location is vacant may result in two nodes 0.

New identifiers for mirrors are strings. They are generated by concatenating string representations of the location of the tree and a counter stored internally. This counter is initialized to zero at tree creation and incremented at each new identifier generated. A possible optimization is to preserve the counters for each location between occupations of that location by an actual facet, giving the generated identifiers lifetimes limited only by counter overflow.

3.3.3 Matlab Bindings

Redefinitions The use of Matlab as a backing application is implemented by redefining methods on Matlab objects. HG operations mostly operate on double-precision handles, so the methods redefined are for class `double`, the built in ordinary number class. The stragglers are redefined in class `char`, ordinary characters. The definitions and some auxilliary functions are all written as M-files.

The method `set` takes a handle and optionally some properties. For cases where the property keys (or values) are not supplied, set has some default behaviours as an enquiry function. If the properties or values are not valid, it ought to throw an error. This behaviour is achieved by passing all of the arguments to the builtin definition of `set`, which operates on the backing instance and supplies some output results, which will be returned.

In the absence of errors, the arguments are then parsed into a list of values and a list of keys. If there are no handles (or no properties), execution returns at this point. Otherwise, the relevant HG objects are queried for a reference to a Java object of class `mirror.Glyph`, which is stored in the `Rune` property of the object[7]. If the query is successful, the result is the facet which this HG object belongs to. The values and keys are then fed into the facet's `set` method, which updates the mirror metadata. Certain metadata are not reflected, and certain other metadata must be reflected as a group[8]. The former are not fed in, the latter are set aside and fed to an alternate syntax for `Glyph.set` which creates and mirrors a multiple `Fact` across the mirror.

The `get` method is redefined similarly. At present, it does not call the `evaluate` method in the glyph, as metadata updates are reflected immediately to the backing object's properties.

Construction is handled by redefining `axes`, `figure`, `image`, `light`, `line`,

[7] Technically, it is stored in a field of the `ApplicationData` property, which is provided by Matlab for such user extensions. This imposes a small performance penalty, but does not otherwise matter.

[8] An example of why is given in section3.3.1.

`patch`, `rectangle`, `surface`, `text`, `uicontrol`, `uimenu`, and `uicontextmenu`. Each new definition calls the built-in constructor, and then checks to see if the new instance should be shared.

The root window, figures and axes are structural. If the user supplies a name (read from the `Tag` property after creation of the backing instance) and the parent instance is shared, then the new child instance is shared also. The parent facet is found from the `Rune` property of the parent instance, and a mirror with the supplied name is `graft`ed onto it. The child facet is inserted into the `Rune` property of the new backing instance, and callbacks in the instance are overridden as described below. The new facet receives an initial copy of the writeable metadata in the backing instance. If the mirror already exists, the graft will involve creating a branch of the instance tree to back the newly grafted facets. This is implemented in the `runify` Matlab function, which recursively builds the HG instances required, from a digest of metadata and object types obtained by traversing the facet subtree.

All other HG objects are half-structural. If the parent instances are shared, the child instance will also be shared. If no tag is supplied for the mirror, one is generated by the root facet for it. All other details are as described above. In section 2.3, non-structural facets propagate along their parent mirrors when they are budded, and disappear from all facets of their parent mirrors when they are deleted. This functionality has not been implemented yet.

Callbacks of HG objects are implemented using the idea of section 2.5. Callbacks are stored as instances of `Executable`, and act as ordinary metadata. Unlike ordinary metadata, they are stored only within the facet. When a HG object is bound as a backing instance to a glyph, the existing callbacks are stored in the facet and the relevant properties replaced with calls to Matlab functions with names like `Callback`. Each of these looks up the relevant callback within the Java facet, and executes what the enquiry returns. By default callbacks execute collaboratively. The enquiry to the facet prompts it to call `evaluate` for that callback key to the last facet that updated it, in a mode that

will execute the callback. The `set.m` implementation is rigged to update only the facet and not the backing instance for callbacks.

Glyph The class `mirror.Glyph` is a subclass of `Rune` specialized for dealing with Matlab backing instances. Instances of `Glyph` possess the following extra state:

- An instance of `com.mathworks.jmi.Matlab`, whose methods it uses to ask Matlab to evaluate function and command calls[9]. They are added to an internal event queue in the backing application instance, and evaluated when idle.
- A double-precision field for the backing instance's handle.
- A string denoting the type of HG object (the constructor name).

The `amend`, `evaluate` and `adopt` implementations in `Rune` delegate operations on the backing instances to protected methods. In `Glyph`, these hooks are overridden to issue the correct Matlab commands - `set` for update, `get` for query, and the appropriate constructors for adoption and budding. This has the consequence that the system must be idle (not executing user code) before these queued commands update the local instance tree.

This means that grafting induced by user Matlab code (such as the construction of a shared object) cannot have queries on aspects of the instance tree interspersed with the facet operations, because the facet method call is executed by the same thread that answers the queries, and deadlock results. As there must be a single thread of control to ensure HG operations execute in the correct order, this introduces significant complications. Notably, the algorithm of `graft` is written in Matlab, and calls Java methods to probe the cutting for structure and metadata, and to supply the backing instances to the cutting in a modified version of `adopt`.

[9] As an optimization, this is actually a class variable.

RootWindow The class `mirror.RootWindow` is a subclass of `Root` specialized to use Matlab as the backing application. If Java permitted multiple inheritance, it would be a subclass of both `Root` and `Glyph`. As it is, the extra fields and reimplementations are direct analogs of those in `Glyph`.

Chapter 4

Empirical Evaluation

4.1 Methodology

After the implementation of Concurrent Matlab stabilized, a short useability trial was run on various machines available to the author. These were:

- A SPARC workstation running Solaris 2.8 with 256 megabytes of RAM and a single 400 MHz processor. This has Matlab 5.3.1, JDK1.2, the prototype Concurrent Matlab over RMI, and SunForum 3.0.
- A generic PC with 16 MB of RAM and a 166Mhz processor. This has Matlab 4.0 and Microsoft NetMeeting installed.
- An IBM SP2, with 128 MB, 266 MHz per node. This has LAM/MPI, Matlab 5.2 and Concurrent Matlab over MPI.

The SP2 nodes are linked by a dedicated high-performance switch, which the LAM/MPI implementation uses in client-to-client mode. All machines are on a common 100-base-T Ethernet which is used by the T.128 applications and the Java RMI stack.

A simple demonstration GUI was written (figure 4.1) consisting of two buttons, a text field, and a line plot showing a randomly generated data set. One button's callback regenerated the data, the other's evaluated the contents of

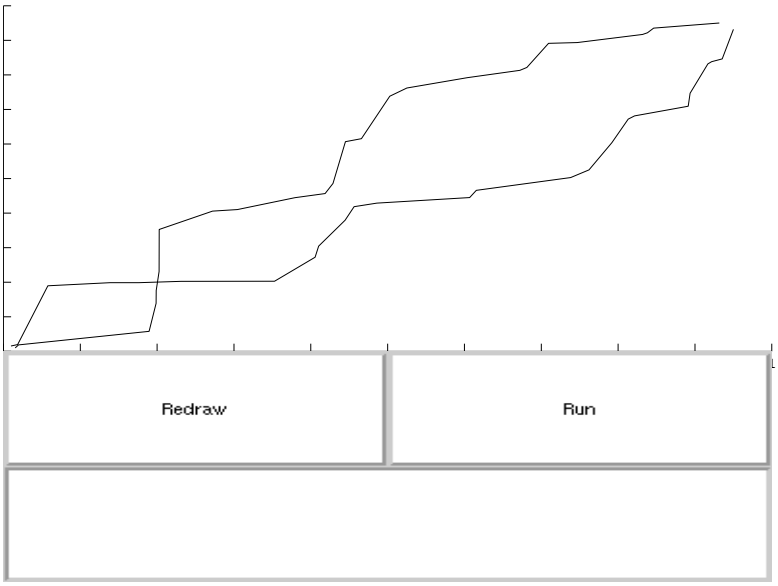

Figure 4.1: Test user interface - screen shot

the text field. The author started a collaborative session, instantiated the GUI, played with the GUI for a while, and then shut down. This process was repeated using various collaboration systems.

4.2 Results

Interactive performance In actual operation, after the shared GUI was created at all places, all systems exhibited a similar response time to user input events (roughly a third of a second). NetMeeting and SunForum were slightly slowed by their implicit floor-control requests all of the time. The Concurrent Matlab prototypes weren't. Concurrent Matlab over RMI exhibited significant lag whilst cloning an entire figure plus contents, and faster speed all of the rest of the time. Concurrent Matlab over MPI exhibited significant lag when invoking callbacks from places not at location 0. Text editing was local in the CM systems, and thus faster than in NetMeeting, which insisted on reflecting each inserted or deleted character.

To truly meet criteria 1, and 2 of the ideal distributed solution, the time

to invoke the actions associated with controls, and the time to see the results reflected, needs to be significantly smaller than the time for the results to be computed. All three systems met this goal.

Distributed Operation All systems managed to span several places comfortably. The primary distinction is in convenience of installation. In descending order of portability:

- Concurrent Matlab over RMI needs a Java virtual machine, and a version of Matlab capable of attaching to it.
- NetMeeting and SunForum are limited to their single platforms and operating systems. By virtue of their adhering to an open standard (T.128), they are able to interoperate with *other* platform-specific T.128 systems, but those will have the same problems.
- Concurrent Matlab in C needs an MPI implementation, a specific compiler and POSIX-compliant library suite, and a local installation of Matlab at each place. This is too specific to be widely distributed.

When in the appropriate environments at each place, all three systems worked reliably over the vanilla Ethernet under conditions of varying load. This is therefore also the order in which they are ranked under criterion 4 of the ideal solution.

Collaborative work Under a T.128 session between SunForum and NetMeeting, one user held the floor and could work, and the other user did not hold the floor and could not work. The user who did not hold the floor could not do local independent work, because these systems usurp the hardware cursor to indicate where the working user is pointing. This group cursor gave more effective awareness of what the active user was doing, and the floor control eliminated consistency issues. Balancing that was the requirement that windows must be unobscured on the machine that shares them, which could not be met conve-

niently. As there was no directory server accessible, sessions were arranged by supplying IP addresses, which was less than convenient.

A Concurrent Matlab over RMI session was held, with both ends on the same machine. Each Matlab instance could be worked with concurrently, and resources not directly shared were free to be used for independent local work. The absence of intrusive floor-control negotiation greatly aided useability, as did the absence of limitations on visibility. Unrestricted collaboration exhibited almost no consistency issues, because the time to mirror changes was so small. Support for group awareness is specialized for Matlab GUI's - the change of a value or invocation of a callback is apparent to all users, but in between there is little indication of what the other user is up to. Sessions were arranged by supplying a session name and contact host name, one step more convenient than SunForum, but not much.

A Concurrent Matlab over MPI session was held, with three instances on three nodes of the SP2. Each instance of the interface could be worked with concurrently, but conflicting requests were not serialized. The various layers of protocol (`mxRequest` serialization down a pipe, queues, MPI, then the Matlab Engine) make the time to reflect changes globally long enough that this is an issue. Support for group awareness is as minimal as the other CM prototype, but not as harmless, because conflicting requests do not result in consistent results across the mirror. Session management is also more awkward than the other systems, requiring the simultaneous coordination of all the users involved.

Of the three systems, the one that best meets criterion 5 of the ideal collaborative solution is Concurrent Matlab, using RMI. The lack of consistency guarantees in both CM prototypes might be awkward, but not quite as awkward as the inability to support flexible, *collaborative* work under the T.128 implementations.

Transparent reusability The test GUI was originally coded in normal Matlab code as a single function to manage the GUI window and take its callbacks.

For SunForum and NetMeeting, it required no modification. For Concurrent Matlab over RMI, it required the addition of tags to a few structural objects. For Concurrent Matlab in C, replacing commands with their MEX-file equivalents required some search-and-replace commands in vi, and the restatement of some of the commands to eliminate convenience syntaxes the MPX library could not parse. The effort involved was trivial in all cases. I believe that in larger cases, tagging of structural HG objects is good software engineering practice, and therefore the effort of porting to Concurrent Matlab over RMI will remain trivial, but the same cannot be said of porting to Concurrent Matlab over MPI. As regards criterion 6, the system that most supports transparently reuseable code is application transparency, very closely followed by the Concurrent Matlab over RMI prototype.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This work is based around the conception of a running model as a set of dynamically changing values. These values are either data, computations that can be run on data (code), or data that is visible and accessible to users (visualization/interface components). In a single place, a model uses these values in a tree, starting from the parent application and adding child procedures, variables, and graphics objects. Each of these kinds of value has a hierarchy which is also a tree. This instance tree represents the computation, and its interaction with the user. In a distributed modelling computation, some of these values will need to be in more than one place at once. Each point of presence, or *facet* of a given value, will need to remain consistent with the other facets. This collection of facets we call a *mirror*. As with normal instances, the various mirrors which overlap at a place contribute their facets to a tree of values. A forest of such *facet trees*, each one shaped by the needs of the user and the computation at each place, can express collaboration among users, concurrency among

computations, and synchronization between both.

The facets of a mirror are obliged to be consistent, but not identical. This difference from simple replicated objects enables a facet to transparently replace an ordinary object in an existing modelling tool, and allow concurrent work on the value within limits. If consistency, synchronization, and communication are handled entirely with the mirror, existing code written in widely-used mathematical languages such as Matlab can be reused easily in collaborative modelling work. The semantics of such a mirror are discussed in depth.

Two specific kinds of facets can be identified in the mathematical language Matlab. *Runes* are facets of a value and a set of metadata - variables. *Glyphs* are runes which are visible to a user - plots, surfaces, and other graphical components. Using these concepts, I have constructed two prototype systems which implement glyphs for this language. These systems attempt to provide a partial solution to the collaborative visualization problem. Their construction drove the generation of theory, and forms the major part of this work. Evaluation of these systems against the relevant criteria for an ideal solution show the ability to meet four of them.

The second prototype, Concurrent Matlab over RMI, represents a novel tool for use with the Matlab language. It uses facet trees as a design pattern for a Java class hierarchy that can closely cooperate with existing Matlab code to obtain transparently collaborative applications. It uses a novel tree-ordered naming scheme for the trees, a refereed algorithm to merge mirrors, and exhibits high responsiveness and transparency in practice. Facet trees are therefore a useful abstraction for reasoning about collaborative visualization in models. It is feasible to map existing code to them generally, in an application-transparent way.

5.2 Future Work

The following is a partial list of research issues to be followed up in future work.

Implementing runes Extending the Java implementation to cover Matlab inline functions, and Matlab workspace variables. At present the aspects of the facet tree framework covering variables and functions are theory with no practice.

Operational transform in trees Researching the applicability of operational transform to tree operations such as insert and delete. When two grafting or pruning operations conflict over a shared mirror, the result is undefined. Serialization is possible using state vectors, but a better solution would be to transform one operation to take account of the other's effect.

Optional locking in mirrors Sosic and Sun [29] have proposed an optional locking scheme, where different users assert ownership of subsets of the shared work. Under this scheme, unlocked portions may be updated concurrently, shared locks may let a subset of places update that section, and exclusive locks implement traditional locking semantics. While this scheme has limited applicability to glyphs, where it would require non-transparent extensions to the user interface for lock management, no such considerations hamper its use by user code (i.e. non-glyph runes). There, conflict avoidance between expensive computations would be much more effective than rolling them back. This scheme can be implemented inside a mirror, by locking subsets of the metadata. Locks can be acquired and released by applying heuristics as facts are exchanged. Research into the correct heuristics will present an interesting challenge.

Visualization as graphics editing Interactive changes to a visualization interface can be considered as an interactive editing session, where the editing operations are initiated from user code. The GRACE project [26] has produced several algorithms and techniques directly applicable to consistency maintenance of shared graphics under unconstrained collaborative editing. The most immediately useful are a set of concrete operational transformations for graphical objects. Another is the concept of compatible and conflicting groups of

operations[27], and an algorithm (MOVIC) for efficiently determining them. When a metadata update arrives at a glyph, it must either accept it as compatible or reject it. A better option than abrupt rejection would be for the glyph to create multiple backing instances, each of which reflects a compatible group of operations applied to it. These compatible groups must be determined by the mirror as a whole, and the MOVIC algorithm provides a means of doing so. Integrating this algorithm into facets raises interesting challenges.

Mathematics as document Matlab regards the local workspace as a set of variables. Other languages, such as Maple, present it as a human-readable document. Previously, operational transform has been applied string-wise to text [28] and object-wise to graphics [26]. I would like to extend it to apply to mathematical expressions. There is a qualitative difference, as expressions are meant to be evaluated into answers, while text and graphics are ends in themselves. Existing document models such as the W3C Document Object Model [8] represent documents as trees of objects, subject to querying and editing in a domain and content-neutral fashion. These trees map naturally to facet trees. A toolset for writing models as pages of mathematics, collaboratively editing, running and visualizing them, would be a worthwhile research goal.

Embedded facets Facets were originally designed as an application-neutral interface to the values that are manipulated in a simulation. There exist a lot of legacy codes which can be used as domain-specific "black boxes". If these codes are equipped with facet behaviour, and the other facets of the mirror are plugged into other, more general modelling tools, then the original codes can be reused more easily and expanded in capability in a general way. This kind of composable simulation [18] is already a topic of active research, mostly centred around the interfaces needed. Facets could be such an interface.

Appendix A

Concurrent Matlab under MPI

This appendix comprises source file listings for the first prototype, Concurrent Matlab using MPI.

A.1 The MPX library

This library expands the MEX and Engine libraries supplied with Matlab, using the LAM/MPI and standard Unix libraries. It is designed to compile as part of either MEX-files (plug-in libraries for Matlab) or into Engine files (standalone executables which interact with Matlab processes). For more information about the Engine and MEX application programming interfaces, please see [15]. For more information about the Message Passing Interface, please see [16].

mpx.h

```
/* file: mpx.h
**
** purpose: header of library of MPI and mxArray widgets for MEX-file use
**
** created: 6/4/2000 by AHD
```

```
*/

/* Redundancy insurance */
#ifndef _MIDGET
#define _MIDGET

/* Private debugging flag */
#ifdef NOT_DEFINED
#define _MPX_DEBUG
#endif

/* Useful headers */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <fcntl.h>
#include <unistd.h>
#include "mpi.h"

/* If it's not a MEX-file, this must be an Engine file */
#ifdef MATLAB_MEX_FILE
   #include "mex.h"
   #define mxPrintf mexPrintf
#else
   #include "engine.h"
   #define mxPrintf printf
#endif

/* Flag bits for a packed mxArray header */
#define LOGICAL_BIT 1
#define COMPLEX_BIT 2
#define EMPTY_BIT 4

/* mxArray functions */
mxArray * mxCreateScalar(double value);
mxArray * mxMakeStructure(mxArray *values[], const char *names[], int nfields);
int mxGetFlags(const mxArray * this);
void * mxPack(const mxArray * this, size_t * bigness);
mxArray * mxUnpack(void * this);

/* Nonblocking file functions */
FILE * propen(const char * path);
FILE * pwopen(const char * path);

/* Miscellaneous definitions */
#define BIG 10240
#define ROOT 0
#define FALSE 0
#define TRUE 1
#define MAYBE 2

#define WORKING 3
```

```
#define LISTENING 4
#define STOPPED 5

/* Standard sizes */
#define mxMAXCMD 1024                /* MATLAB engine command buffer */
#define mxMAXRQ        512               /* Entries in a request queue */
#define mxMAXOUT 10240               /* MATLAB engine output buffer */
#define mxMAXHDR 10                /* Scalar fields in an mxArray header */
#define mxMAXCBACKS 10               /* Number of callbacks registered per object */

/* Kinds of mxRequests */
#define NO_REQUEST 0
#define SET_REQUEST 1
#define GET_REQUEST 2
#define CREATE_REQUEST 3
#define DESTROY_REQUEST 4
#define CALLBACK_REQUEST 5
#define EVALUATE_REQUEST 6

/* Default values for the status fields of mxRequests */
#define REQUEST_INVALID -1
#define REQUEST_VALID 1

/* Kinds of callback */
#define CALL_INVALID 0               /* For future use */
#define CALL_PRIMARY 1
#define CALL_BUTTONUP 2
#define CALL_BUTTONDWN 3
#define CALL_MOTION 4
#define CALL_DELETE 5
#define CALL_RESIZE 6
#define CALL_CHANGE 7


/* Structure representing a Handle Graphics request */
typedef struct {
   int kind;                    /* What kind of request this is */
   int status;                        /* Further detail on the above */
   char * words;                /* Associated name */
   mxArray * array;             /* Associated Matlab value */
   mxArray * object;            /* Associated Matlab handles (at root) */
   double parent;               /* Parent of object (at root)  */
   int origin;                        /* Which node originated this request */
   int location;                /* Which node the request is in now */
   int references;                /* Reference count */
} mxRequest;

/* Structure representing a thread-safe FIFO queue of mxRequests */
typedef struct {
   mxRequest * contents[mxMAXRQ];        /* List of entries in the queue */
   int head;                               /* Index of head of queue */
   int tail;                               /* Index of tail of queue */
   int count;                               /* Population of queue */
   const char * name;                       /* Name of queue */
```

```
   pthread_mutex_t gate;               /* Gatekeeper mutex */
} mxQueue;

/* Functions to play with queues of requests */
mxQueue * mxInitQueue(const char * name);
void mxFinalizeQueue(mxQueue *q);
void mxEnqueue(mxRequest * this, mxQueue *q);
mxRequest * mxDequeue(mxQueue *q);
int mxQueueCount(mxQueue *q);

/* Functions to play with requests */
void mxDestroyRequest(mxRequest * this);
mxRequest * mxCreateRequest(int kind, int origin, int here);
mxRequest * mxStructureToRequest(mxArray * box, int kind, int here);
mxArray * mxRequestToStructure(mxRequest * this);
int mxIsLocal(mxRequest * this);

/* A magic bullet is a request to destroy the root object (0.0) */
mxRequest * mxCreateMagicBullet(int here);
int mxIsMagicBullet(mxRequest * this);

/* Functions to shift requests around via MPI */
int mxSendRequest(int dest, MPI_Comm comm, mxRequest * this, int src);
mxRequest * mxRecvRequest(int source, MPI_Comm comm, int here);

/* Functions to shift requests around via pipes */
int mxWriteRequest(mxRequest * this, int pipe);
mxRequest * mxReadRequest(int pipe, int here);

/* Graphics status values */
#define STATUS_NOMINAL 0
#define STATUS_INVALID (-1)

/* An object handle guaranteed to be invalid */
#define INVALID_HANDLE (-1.0)

/* Structure representing a Handle Graphics object */
typedef struct _object {
   double handle;              /* What this instance of Matlab calls it */
   double rootID;              /* What the root calls it */
   double rootParent;          /* What the root calls its' parent */
   char * callbacks[mxMAXCBACKS];    /* Collection of callbacks */
   int status;                 /* What are we doing to it? */
   struct _object * next;      /* Away from head of list */
   struct _object * previous;  /* Towards head of list */
} mxObject;

/* Structure representing a thread-safe dictionary of mxObjects */
typedef struct {
   char * name;                      /* What it's called */
   mxObject * start;               /* Head of the list */
   int count;                       /* Population of the list */
   pthread_mutex_t * gate;        /* Gatekeeper mutex */
} mxDictionary;
```

```
/* Functions to play with mxObjects */
mxObject * mxCreateObject(double h, double r, double p);
void mxDestroyObject(mxObject * this);
void mxSetObjectCallback(mxObject * this, int which, const char * call);
const char * mxGetObjectCallback(mxObject * this, int which);
void mxSetObjectStatus(mxObject * this, int quo);
int mxGetObjectStatus(mxObject * this);

/* Functions to play with mxDictionaries */
mxDictionary * mxCreateDictionary(char * name);
void mxDestroyDictionary(mxDictionary * victim);
void mxAddObject(mxDictionary * those, mxObject * this);
void mxRemoveObject(mxDictionary * those, mxObject * this);
mxObject * mxGetObjectByRoot(mxDictionary * those, double r);
mxObject * mxGetObjectByParent(mxDictionary * those, double p);
mxObject * mxGetObjectByHandle(mxDictionary * those, double h);
mxArray * mxGetLocalHandles(const mxArray * handles, mxDictionary * those);
mxArray * mxGetRootHandles(const mxArray * handles, mxDictionary * those);
int mxExists(const mxArray * handles, mxDictionary * those);
mxArray * mxCensorProperties(const mxArray * old, mxDictionary * those);
void mxListDictionary(mxDictionary * those, int here);
void mxUpdateDictionary(mxDictionary * those, mxRequest * r, mxArray * h);

#ifdef MATLAB_MEX_FILE

/* MEX-file argument parsers */
int mexParseOne(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseC(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseXY(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseXYC(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseXYT(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseXYZ(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseXYZT(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
int mexParseXYZC(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]);
#else

/* Function to put an mxRequest to an Engine using a mxDictionary */
char * engRequest(Engine * e, mxRequest * this, mxDictionary * those);

#endif

#endif
```

mpx.c

```
/* file: mpx.c
**
** purpose: library of MPI and mxArray widgets for MEX-file use
**
** created: 6/4/2000 by AHD
*/

#include "mpx.h"

/* A note on conventions:
   * "RCF" is short for "Reality Check Failed". It denotes Bad Things.
   * mxArrays are a Matlab-defined opaque type. Constant pointers to them
     denote arrays that must be copied before being changed or used as an
     ingredient in a larger array; Matlab checks this thoroughly and flags
     exceptions to it. When the checking fails, strange and subtle errors
     proliferate, so don't cast them to volatiles, use mxDuplicateArray.
   * Whilst all of the collective data types are thread-safe, the objects
     they point to are not. mxObjects and mxRequests should never be fiddled
     with whilst part of a mxDictionary or a mxQueue. Interface functions to
     guarantee this will be added Real Soon Now.
   * The adjective "jobbing" refers to buffer or temporary variables which
     are used for multiple purposes. */


/* Dual to mxGetScalar.
   Returns a pointer to a real, 1x1 mxArray with the value specified,
   or NULL on allocation failure.  */
mxArray * mxCreateScalar(double value) {
   mxArray * that = mxCreateDoubleMatrix(1,1,mxREAL);
   if (that) (mxGetPr(that))[0] = value;
   return that;
}

/* Simplified variant of mxCreateStruct.
   Returns a pointer to a 1-cell structure mxArray with the field names and
   values specified, or NULL on allocation failure. */
mxArray * mxMakeStructure(mxArray *values[], const char *names[], int nfields) {
   mxArray * array = NULL;
   int i;
   if (array = mxCreateStructMatrix(1,1, nfields, names)) {
      for (i=0; i<nfields; i++) mxSetField(array, 0, names[i], values[i]);
   }
   return array;
}

/* Examines a given mxArray and returns a flag byte for use in a packed
   header. The bits are defined in mpx.h. "this" must be a valid pointer.  */
int mxGetFlags(const mxArray * this) {
   return (mxIsLogical(this) ? LOGICAL_BIT : 0)
        + (mxIsComplex(this) ? COMPLEX_BIT : 0)
        + (mxIsEmpty(this) ? EMPTY_BIT : 0);
```

```
}


/* This packs one cell of a structure mxArray into a contiguous binary object.
   It returns a pointer to a concatenated copy of all the relevant fields of
   that cell. The second parameter gives the extent in bytes. The third
   gives the index of the mxArray cell to copy. It calls mxPack recursively
   to pack the field values. The first parameter must be a valid mxArray
   pointer. */
void * mxPackOneStruct(const mxArray *this, size_t *l, int a) {
   void * places[256];        /* List of addresses to copy from */
   size_t sizes[256];        /* Number of bytes to copy from each */
   int parts=0;                /* Length of these two lists */
   char * cursor;        /* Current copying point */
   size_t total=0, i;         /* Tallying variables */
   int fields;                /* Number of fields in this structure */
   int last;                /* Last header item */
   char * blob;                /* End result */

   /* Lay in header */
   fields = mxGetNumberOfFields(this);        /* Number of fields */
   places[0]=&fields; sizes[0]=sizeof(int); parts++;
   places[parts]=&(sizes[parts+1]);         /* Field offsets */
   sizes[parts]=fields*sizeof(size_t); last = parts; parts++;

   /* Lay in fields */
   for (i=0; i<fields; i++) {
      mxArray * that = mxGetFieldByNumber(this, a, i);
      places[parts] = mxPack(that, &(sizes[parts])); parts++;
   }

   /* Allocate some memory and compile everything */
   for (i=0, total=0; i<parts; i++) total += sizes[i];
   blob = mxMalloc(total);
   for (i=0,cursor=(char *)blob; i<parts; cursor+=sizes[i], i++) {
      memcpy((void *)cursor, places[i], sizes[i]);
      if (i>last) mxFree(places[i]);
   }

   if (l) *l = total;        /* Return bytes created */
   return blob;                /* Return package */
}

/* Dual to mxPackOneStruct.
   The first parameter is a binary object containing the field values for one
   cell of the third parameter, a structure mxArray. The fourth parameter
   denotes which cell of the structure to fill.
   This function decodes the header, calls mxUnpack to create the values,
   and puts them in the right place.
   The format is self-delimiting, however mxUnpack needs to know how many
   bytes of the binary object were consumed. This is returned in the
   second parameter. */
void mxUnpackOneStruct(char *blob, size_t *offset, mxArray * term, int index) {
   mxArray * that;                /* Value under consideration */
```

```
   int fields;                        /* Number of fields to unpack */
   size_t * offsets;                 /* Field offsets */
   char * cursor;                 /* Current unwinding point */
   int i;                         /* Tallying variable */

   /* Read off number of fields and offsets within the blob */
   cursor = blob;
   fields = *((int *)cursor); cursor += sizeof(int);
   offsets = (size_t *)cursor; cursor += fields*sizeof(size_t);

   /* Unpack the fields and put them into the matrix */
   for (i=0; i<fields; i++) {
      that = mxUnpack((void *)cursor);
      mxSetFieldByNumber(term, index, i, that);
      cursor += offsets[i];
   }
   if (offset) (*offset) = cursor-blob; /* Return bytes consumed */
}


/* Takes an arbitrary mxArray as its' first parameter, and creates a
   contiguous binary object that represents it. It returns a pointer to the
   object, or NULL if the mxArray cannot be packed. It returns the size of
   the object in bytes via the second parameter.
   Currently supported types of mxArray are full double-precision numeric
   arrays, character arrays (Matlab strings) and structure arrays. */
void * mxPack(const mxArray * this, size_t * bigness) {
   mxClassID kind;        /* What class of array this is */
   int flag;                /* Flag byte */
   int dims;                /* Number of dimensions */
   const int * dlist;        /* List of dimensions */
   int fields, nelem;        /* Number of fields and elements */
   char ** flist;        /* List of field names, if any */

   int parts=0;                /* Number of things to copy */
   void * places[256];        /* List of addresses to copy from */
   size_t sizes[256];        /* Number of bytes to copy from each */
   char wild[256];        /* TRUE if that address needs to be freed */
   void * blob;                /* The end result */
   char * cursor;        /* Current copying point */
   size_t total=0, i;        /* Tallying variables */

   bzero(wild, 256*sizeof(char));        /* Flag all addresses tame */

   /* Diagnose array, prepare header */
   kind = mxGetClassID(this);         /* What kind? */
   places[0]=&kind; sizes[0]=sizeof(mxClassID); parts++;
   flag = mxGetFlags(this);        /* What flags? */
   places[parts]=&flag; sizes[parts]=sizeof(int); parts++;
   places[parts]=(void *)mxGetName(this); /* What array name? */
   sizes[parts]=(mxMAXNAM)*sizeof(char); parts++;
   dims = mxGetNumberOfDimensions(this); /* How many dimensions? */
   places[parts]=&dims; sizes[parts]=sizeof(int); parts++;
   dlist = mxGetDimensions(this);        /* Dimension details */
```

```
   places[parts]=(void *)dlist; sizes[parts]=dims*sizeof(int); parts++;

   /* Lay in data */
   if (mxIsEmpty(this)) {
      /* No data to be copied */
   } else if (kind == mxDOUBLE_CLASS) {        /* Double precision tensor */
      places[parts]=(void *)mxGetPr(this); /* Real part */
      sizes[parts]=mxGetNumberOfElements(this)*sizeof(double); parts++;
      if (mxIsComplex(this)) {
         places[parts]=(void *)mxGetPi(this); /* Optional imaginary part */
         sizes[parts]=mxGetNumberOfElements(this)*sizeof(double); parts++;
      }
   } else if (kind == mxCHAR_CLASS) {        /* Unicode character array */
      places[parts]=mxGetData(this);
      sizes[parts]=mxGetNumberOfElements(this)*sizeof(mxChar); parts++;
   } else if (kind == mxSTRUCT_CLASS) {
      fields = mxGetNumberOfFields(this);
      nelem = mxGetNumberOfElements(this);
      places[parts] = (char *)(&fields); sizes[parts] = sizeof(int); parts++;
      for (i=0; i<fields; i++) {
         places[parts] = (void *)mxGetFieldNameByNumber(this, i);
         sizes[parts] = (strlen(places[parts])+1)*sizeof(char); parts++;
      }
      for (i=0; i<nelem; i++) {
         places[parts] = mxPackOneStruct(this, &(sizes[parts]),i);
         wild[parts] = TRUE; parts++;
      }
   } else {
      fprintf(stderr, "%s isn't something I can pack.\n", mxGetName(this));
      return NULL;
   }

   /* Allocate some memory and compile everything. */
   for (i=0, total=0; i<parts; i++) total += sizes[i];
   blob = mxMalloc(total);

   for (i=0,cursor = (char *) blob; i<parts; cursor+=sizes[i],i++) {
      memcpy((void *) cursor, places[i], sizes[i]);        /* Copy */
      if (wild[i]) mxFree(places[i]);                        /* Free */
   }

   if (bigness) (*bigness) = total;        /* Return size of the blob */
   return blob;                                /* Return location of the blob */
}

/* Takes a binary object created by mxPack and recreates the corresponding
   mxArray. It returns a pointer to the mxArray created. If the object cannot
   be unpacked, it returns a pointer to the scalar 42. */
mxArray * mxUnpack(void * this) {
   mxClassID kind;        /* What class of array this is */
   int flag;                /* Flag bits */
   char * name;                /* Name of the matrix */
   mxArray * result;        /* End product */
   char * cursor;        /* Where were we? */
```

```
int dims;                /* Number of dimensions */
int * dlist;               /* List of dimensions */
char isComplex;        /* Is it complex? */
char isLogical;        /* Is it a logical array? */
char isEmpty;        /* Is it an empty array? */
int nelem, fields;        /* Number of dimensions and fields */
char ** flist;        /* List of field names, if any */
int i;                /* Jobbing counter */

/* Parse the header */
cursor = (char *) this;
kind = * (mxClassID *)(cursor); cursor += sizeof(mxClassID);
flag = * (int *)(cursor); cursor += sizeof(int);
isComplex = flag & COMPLEX_BIT; isLogical = flag & LOGICAL_BIT;
isEmpty = flag & EMPTY_BIT;
name = (char *)(cursor); cursor += mxMAXNAM*sizeof(char);

/* Grab the dimensions, unpack the data */
if (kind == mxDOUBLE_CLASS) {                /* Double precision tensor */
   dims = * (int *)(cursor); cursor += sizeof(int);
   dlist = (int *)(cursor); cursor += dims * sizeof(int);
   result = mxCreateNumericArray(dims, dlist, kind,
            isComplex ? mxCOMPLEX : mxREAL);
   if (isEmpty) {
      /* Leave it empty */
   } else {         /* Copy in real components of data */
      memcpy(mxGetData(result),(void *)cursor,
             mxGetNumberOfElements(result)*sizeof(double));
      cursor += mxGetNumberOfElements(result)*sizeof(double);
      if (isComplex) {        /* Copy imaginary components, if any */
         memcpy(mxGetImagData(result),(void *)cursor,
                mxGetNumberOfElements(result)*sizeof(double));
         cursor += mxGetNumberOfElements(result)*sizeof(double);
      }
   }
} else if (kind == mxCHAR_CLASS) {                /* Character array */
   dims = * (int *)(cursor); cursor += sizeof(int);
   dlist = (int *)(cursor); cursor += dims * sizeof(int);
   result = mxCreateCharArray(dims, dlist);
   if (isEmpty) {
      /* Leave it empty */
   } else {         /* Copy character data in */
      memcpy(mxGetData(result),(void *)cursor,
             mxGetNumberOfElements(result)*sizeof(mxChar));
      cursor += mxGetNumberOfElements(result)*sizeof(mxChar);
   }
} else if (kind == mxSTRUCT_CLASS) {                /* Structure array */
   dims = * (int *)(cursor); cursor += sizeof(int);
   dlist = (int *)(cursor); cursor += dims * sizeof(int);
   fields = *(int *)(cursor); cursor += sizeof(int);
   flist = mxCalloc(fields, sizeof(char *));
   for (i=0; i<fields; i++) {
      flist[i] = strdup(cursor);
      cursor += (strlen(flist[i]) + 1);
```

```
      }
      result = mxCreateStructArray(dims, dlist, fields, (const char **)flist);
      if (isEmpty) {
         /* Leave it empty */
      } else {         /* Unpack each field */
         nelem = mxGetNumberOfElements(result);
         for (i=0; i < nelem; i++) {
             size_t lump; /* Bytes for this cell of the array */
             mxUnpackOneStruct(cursor, &lump, result, i);
             cursor += lump;
         }
      }
   } else {
      printf("You want me to unpack that?\n");
      result = mxCreateScalar(42);
   }
   mxSetName(result, name);
   return result;
}

/* Open a FIFO for reading in nonblocking mode */
FILE * propen(const char * path) {
   return fdopen(open(path, (O_RDONLY|O_NDELAY)),"rb");
}

/* Open a FIFO for writing in nonblocking mode */
FILE * pwopen(const char * path) {
   return fdopen(open(path, (O_WRONLY|O_NDELAY)),"wb");
}

/* Current implementation of an mxQueue is a circular buffer of size
   mxMAXRQ. The head points to the first element. The tail points to the
   slot where the next element will go. count = head-tail, modulo mxMAXRQ.
   head = tail implies an empty queue. The queue holds mxMAXRQ-1 requests.*/

/* Creates an empty mxQueue.
   Returns a pointer to it, or NULL on allocation failure.  */
mxQueue * mxInitQueue(const char * name) {
   mxQueue * q;
   if (q = mxCalloc(1,sizeof(mxQueue))) {
      q->head = q->tail = q->count = 0; q->name = strdup(name);
      if (pthread_mutex_init(&(q->gate),NULL)) {
         mxPrintf("RCF: %s mutex isn't. \n", name); mxFree(q);
      }
   } else {
      mxPrintf("RCF: %s queue isn't.\n");
   }
   return q;
}
/* Destroys an mxQueue.
   Dequeues and destroys any remaining entries, destroys mutex and frees it. */
void mxFinalizeQueue(mxQueue *q) {
   if (q) {
      mxRequest * i;
```

```
      while (i = mxDequeue(q)) mxDestroyRequest(i);        /* Empty the queue */
      if (pthread_mutex_destroy(&(q->gate)))         /* Destroy mutex */
         mxPrintf("RCF: Can't kill mutex on queue %s.\n", q->name);
      mxFree(q);
   } else {
      mxPrintf("RCF: Destroying a null queue.\n");
   }
}
/* Enqueues an mxRequest on an mxQueue. */
void mxEnqueue(mxRequest * this, mxQueue *q) {
   if (pthread_mutex_lock(&(q->gate))) {
      mxPrintf("RCF: Lock failure on queue %s.\n", q->name);
   } else {
      if (this) {
         int next = (q->tail + 1) % mxMAXRQ;        /* New tail position */
         if (next != q->head) {
            q->contents[q->tail] = this; q->count++; q->tail = next;
            this->references++;                          /* One more reference */
            #ifdef _MPX_DEBUG
            mxPrintf("[%d] Enqueueing a type %d at #%d in %s.\n",
                     this->location, this->kind, next, q->name);
            #endif
         } else {
            mxPrintf("RCF: Full %s queue.\n");
         }
      } else {
         mxPrintf("RCF: Enqueueing null request on %s.\n",q->name);
      }
      pthread_mutex_unlock(&(q->gate));
   }
}
/* Dequeues a mxRequest from an mxQueue.
   Returns a pointer to the mxRequest, or NULL if the queue is empty. */
mxRequest * mxDequeue(mxQueue *q) {
   mxRequest * result = NULL;
   if (pthread_mutex_lock(&(q->gate))) {
      mxPrintf("RCF: Lock failure on queue %s.\n", q->name);
   } else {
      if (q->count) {        /* If requests in queue */
         int next = (q->head + 1) % mxMAXRQ;        /* New head of queue */
         result = q->contents[q->head]; q->contents[q->head] = NULL;
         q->head = next; q->count--; result->references--;
         #ifdef _MPX_DEBUG
         mxPrintf("[%d] Dequeueing a type %d at #%d from %s.\n",
                  result->location, result->kind, next, q->name);
         #endif
      }
      pthread_mutex_unlock(&(q->gate));
   }
   return result;
}
/* Returns the number of mxRequests still in this queue.
   Shows an empty queue on failure. */
int mxQueueCount(mxQueue *q) {
```

```
   int result = 0;
   if (pthread_mutex_lock(&(q->gate))) {
      mxPrintf("RCF: Lock failure on queue %s.\n", q->name);
   } else {
      result = q->count;
      pthread_mutex_unlock(&(q->gate));
   }
   return result;
}
/* Destroys an mxRequest. If the reference count is positive, it does
   nothing. Otherwise, it frees all the fields. */
void mxDestroyRequest(mxRequest * this) {
   if (this) {
      if (this->references > 0) {
         /* mxPrintf("RCF: Destroying a referenced request.\n"); */
      } else {
         if (this->words) mxFree(this->words);
         if (this->object) mxDestroyArray(this->object);
         if (this->array) mxDestroyArray(this->array);
         mxFree(this);
      }
   } else {
      mxPrintf("RCF: Destroying a null request.\n");
   }
}

/* Creates a request with a specified kind, origin and location.
   Returns a pointer to that request or NULL on allocation failure. */
mxRequest * mxCreateRequest(int kind, int origin, int here) {
   mxRequest * r;
   if (r = mxCalloc(1, sizeof(mxRequest))) {
      r->object = r->array = NULL; r->words = NULL;
      r->origin = origin; r->location = here; r->references = 0;
      r->kind = kind; r->status = REQUEST_VALID;
   } else {
      mxPrintf("RCF: No memory left for a request.\n");
   }
   return r;
}
/* Creates a magic bullet.
   A magic bullet is a request to destroy the root window (handle 0.0).
   When a daemon sees this, it is supposed to commit suicide.
   Returns a pointer to the bullet, or NULL on allocation failure. */
mxRequest * mxCreateMagicBullet(int here) {
   mxRequest * r =  mxCreateRequest(DESTROY_REQUEST, here, here);
   if (r) r->object = mxCreateScalar(0.0);
   return r;
}
/* Returns TRUE if this request is a magic bullet */
int mxIsMagicBullet(mxRequest * this) {
   if ((this)&&(this->kind == DESTROY_REQUEST)) {
      if (mxGetScalar(this->object)==0.0) return TRUE;
   }
   return FALSE;
```

```
}
/* Returns TRUE if this mxRequest's location matches its' origin. */
int mxIsLocal(mxRequest * this) {
   if (this) {
      if (this->location == this->origin) return TRUE;
   }
   return FALSE;
}


/* Packs and sends an mxRequest via MPI.
   The parameters are destination, MPI communicator, mxRequest to send and
   the sender's rank on that communicator, for backward compatibility.
   The mxRequest is converted first into an equivalent mxArray, and then
   into a binary object, which is sent as a single MPI message. The
   mxRequest kind field is taken as the tag for the message. This protocol
   only works between processes running on compatible platforms. */
int mxSendRequest(int dest, MPI_Comm comm, mxRequest * this, int src) {
   mxArray * box;                /* Structure array with the request in it */
   void * parcel;                /* Structure array packed for shipping */
   size_t broadness;                /* Size of the parcel */
   int code;                        /* Return code from MPI send */

   /* Compile, pack for shipping, and send */
   box = mxRequestToStructure(this);
   parcel = mxPack(box, &broadness);
   code = MPI_Send(parcel, broadness, MPI_BYTE, dest, this->kind, comm);
   /* Untangle all the pointers and kill off the copies */
   mxFree(parcel);        /* Perhaps cache this? */
   mxDestroyArray(box);        /* All fields are just duplicates */

   #ifdef _MPX_DEBUG
   mxPrintf("[%d] Request sent in %d bytes.\n", this->location, broadness);
   #endif

   return code;
}

/* Makes a 1x1 structure array with equivalent fields to a given mxRequest.
   The "array", "words", and "object" fields are optional.  Returns a pointer
   to the resulting mxArray, or NULL on allocation failure. */
mxArray * mxRequestToStructure(mxRequest * this) {
   const char * names[mxMAXHDR];/* Fields in structure */
   mxArray * values[mxMAXHDR];        /* Values in structure */
   mxArray * scalars;                /* Scalar values */
   double * scdata;                /* Its' real elements */
   int nfields, i;                /* Number of fields, counter over them */
   mxArray * box;                /* Structure array with the above in it */

   scalars = mxCreateDoubleMatrix(1, mxMAXHDR, mxREAL);        /* Scalar fields */
   scdata = mxGetPr(scalars);
   scdata[0] = (double)(this->kind); scdata[1] = (double)(this->status);
   scdata[2] = (double)(this->origin); scdata[3] = this->parent;
   names[0] = "scalars"; values[0] = scalars; nfields = 1;
```

```
   #ifdef _MPX_DEBUG
   mxPrintf("[%d] Encrypting a type %d.\n", this->location, this->kind);
   #endif

   if (this->object) {        /* Add object handles, if any */
      names[nfields] = "object";
      values[nfields] = mxDuplicateArray(this->object); nfields++;
      #ifdef _MPX_DEBUG
      mxPrintf("[%d] Outgoing object is %g.\n", this->location,
         mxGetScalar(this->object));
      #endif

   }
   if (this->array) {        /* Add values, if any */
      names[nfields] = "array";
      values[nfields] = mxDuplicateArray(this->array); nfields++;
   }
   if (this->words) {        /* Add prose if any */
      names[nfields] = "words";
      values[nfields] = mxCreateString(this->words); nfields++;
      #ifdef _MPX_DEBUG
      mxPrintf("[%d] Outgoing words are %s.\n", this->location, this->words);
      #endif
   }
   box = mxMakeStructure(values, names,nfields);
   return box;
}

/* Receives an mxRequest from a given source on an MPI communicator
   without blocking. The mxRequest must be sent via mxSendRequest.
   If there is a pending message, this function receives and unpacks it.
   The request is created with the location supplied in the third parameter.
   A pointer to the mxRequest is returned on success, or NULL on failure. */
mxRequest * mxRecvRequest(int source, MPI_Comm comm, int here) {
   void * parcel;               /* Raw message */
   size_t broadness;               /* Its' size */
   mxArray * box, * field;       /* Unpacked parcel */
   double * scalars;               /* Scalar fields */
   int kind, arePending, s;       /* Probe results */
   MPI_Status stat;               /* Jobbing status indicator */
   int origin;                        /* Origin of the request */
   mxRequest * this = NULL;       /* The end result */

   MPI_Iprobe(source, MPI_ANY_TAG, comm, &arePending, &stat);
   if (arePending) {
      kind = stat.MPI_TAG; MPI_Get_count(&stat, MPI_BYTE, &s);
      broadness = (size_t)s; parcel = mxMalloc(broadness);
      MPI_Recv(parcel, s, MPI_BYTE, source, kind, comm, &stat);
      box = mxUnpack(parcel); mxFree(parcel);        /* Perhaps cache this? */
      this = mxStructureToRequest(box, kind, here);
      mxDestroyArray(box);        /* Providing only copies are destroyed */
      #ifdef _MPX_DEBUG
      if (this) mxPrintf("[%d] Request received in %d bytes\n", here,broadness);
```

```
      #endif
   }
   return this;
}

/* Dual to mxRequestToStructure. From the fields of the first cell of a
   structure mxArray, it creates an equivalent mxRequest and returns a
   pointer to it. The mxRequest is created with the location specified by
   the third parameter. */
mxRequest * mxStructureToRequest(mxArray * box, int kind, int here) {
   mxRequest * this;               /* The end result */
   double * scalars;               /* The scalar fields */
   mxArray *field;               /* The tensor fields */
   scalars = mxGetPr(mxGetField(box, 0, "scalars"));
   this = mxCreateRequest(kind, here, here);
   this->status = (int)(scalars[1]); this->origin = (int)(scalars[2]);
   this->parent = scalars[3]; this->location = here;

   #ifdef _MPX_DEBUG
   mxPrintf("[%d] Decrypting a type %d.\n", here, kind);
   #endif
   if (field = mxGetField(box, 0, "words")) {
      this->words = mxArrayToString(field);
      #ifdef _MPX_DEBUG
      mxPrintf("[%d] Words are %s.\n", here, this->words);
      #endif
   }
   if (field = mxGetField(box, 0, "array")) {
      this->array = mxDuplicateArray(field);
   }
   if (field = mxGetField(box, 0, "object")) {
      this->object = mxDuplicateArray(field);
      #ifdef _MPX_DEBUG
      mxPrintf("[%d] Object is %g.\n", here, mxGetScalar(this->object));
      #endif
   }
   return this;
}


/* Writes an mxRequest to a given file descriptor, usually a pipe.
   First the request kind field, then the size of the packed mxRequest, then
   the request itself are written. Returns the result of the first write(). */
int mxWriteRequest(mxRequest * this, int pipe) {
   mxArray * box;                /* Buffer structure */
   void * package;                /* Jobbing buffer variable */
   size_t bigness;                /* Elements to write */
   ssize_t written;                /* Elements actually written */
   int kind = this->kind;        /* Request tag */

   written = write(pipe, (void *)(&kind), sizeof(int));        /* Request */
   if (written > 0) {        /* If pipe is functional */
      box = mxRequestToStructure(this);        /* Turn into mxArray */
      package = mxPack(box, &bigness);        /* Turn mxArray into binary */
```

```
      write(pipe, (void *)(&bigness), sizeof(size_t));/* Write size */
      written = write(pipe, package, bigness);        /* Write request */
      #ifdef _MPX_DEBUG
      mxPrintf("Request written in %d/%d bytes.\n", written, bigness);
      #endif
      mxFree(package); mxDestroyArray(box); /* Cache? */
   } else {
      mxPrintf("RCF: pipe %d full.\n", pipe);
   }
   return written;
}


/* Reads an mxRequest from a given file descriptor, usually a pipe.
   The mxRequest must have been sent with mxWriteRequest.
   The first read() is nonblocking. This function returns a pointer to
   a valid mxRequest if one was in the pipe, otherwise NULL. */

mxRequest * mxReadRequest(int pipe, int here) {
   mxRequest * this;                /* Request under construction */
   mxArray * box;                /* The wrapping for it */
   void * package;                /* Jobbing buffer variable */
   size_t bigness;                /* Elements to write */
   int kind, status, origin;        /* Request fields */
   ssize_t readcount;                /* Bytes actually read */

   readcount = read(pipe, (void *)(&kind), sizeof(int)); /* Kind of request */
   if (readcount > 0) {
      readcount = read(pipe, (void *)(&bigness), sizeof(size_t));
      if (readcount > 0) {
         package = mxMalloc(bigness);
         readcount = read(pipe, package, bigness);
         if (readcount > 0) {
            box = mxUnpack(package); mxFree(package);
            this = mxStructureToRequest(box, kind, here);
            #ifdef _MPX_DEBUG
            mxPrintf("Request read with %d/%d bytes\n", readcount, bigness);
            #endif
         } else {
            mxPrintf("RCF: read() for package returned %d\n", readcount);
         }
      } else {
         mxPrintf("RCF: read() for package size returned %d\n", readcount);
      }
   } else {
      this = NULL;        /* No request queued */
      if (readcount < 0) {
         mxPrintf("RCF: read() for tag returned %d\n", readcount);
      }
   }
   return this;
}

/* Creates a blank mxObject with no callbacks.
```

```
   The parameters are the local handle, the handle at the root node, and
   the handle of the parent at the root node, in order. The mxObject created
   has those details and no callbacks registered. On success, a pointer to
   the mxObject is returned. On failure, NULL is returned. */
mxObject * mxCreateObject(double h, double r, double p) {
   mxObject * this;
   if (this = mxCalloc(1, sizeof(mxObject))) {
      #ifdef _MPX_DEBUG
      mxPrintf("[x] Object %g, child of %g, is locally %g. \n", h, p, r);
      #endif
      this->handle = h; this->rootID = r; this->rootParent = p;
   } else {
      mxPrintf("RCF: no memory for a new object.\n");
   }
   return this;
}
/* Destroys an mxObject and frees all components. */
void mxDestroyObject(mxObject * this) {
   int i;
   if (this) {
      for (i=0; i<mxMAXCBACKS; i++) mxFree(this->callbacks[i]);
      mxFree(this);
   } else {
      mxPrintf("RCF: destroying a null object.\n");
   }
}
/* Updates a callback of an mxObject.
   If a previous callback string is found, it is freed. */
void mxSetObjectCallback(mxObject * this, int which, const char * call) {
   if (this) {
      #ifdef _MPX_DEBUG
      mxPrintf("[x] Callback %d of %g is [%s]\n", which, this->handle, call);
      #endif
      if (this->callbacks[which]) mxFree(this->callbacks[which]);
      this->callbacks[which] = strdup(call);
   } else {
      mxPrintf("RCF: setting callback of a null object.\n");
   }
}
/* Retrieves a callback of an mxObject; returns NULL if none */
const char * mxGetObjectCallback(mxObject * this, int which) {
   if (this) {
      return this->callbacks[which];
   } else {
      mxPrintf("RCF: getting callback of a null object.\n"); return NULL;
   }
}
/* Sets the status field of an mxObject */
void mxSetObjectStatus(mxObject * this, int quo) {
   if (this) {
      this->status = quo;
   } else {
      mxPrintf("RCF: setting status of a null object.\n");
   }
```

```
}
/* Returns the status field of an mxObject, or STATUS_INVALID on failure */
int mxGetObjectStatus(mxObject * this) {
   if (this) {
      return this->status;
   } else {
      mxPrintf("RCF: getting status of a null object.\n");
      return STATUS_INVALID;
   }
}
/* Creates an empty mxDictionary. Returns a pointer to it on success, or
   NULL on failure. Requires a name; this is used in debugging traces.*/
mxDictionary * mxCreateDictionary(char * name) {
   mxDictionary * d;
   if (d = mxCalloc(1, sizeof(mxDictionary))) {
      d->name = strdup(name); d->start = NULL; d->count = 0;
      d->gate = mxMalloc(sizeof(pthread_mutex_t));
      if (pthread_mutex_init(d->gate,NULL)) {
         mxPrintf("RCF: lock for %s dictionary isn't.\n", name);
         mxFree(d->gate); mxFree(d); d = NULL;
      }
   } else {
      mxPrintf("RCF: no memory for %s dictionary.\n");
   }
   return d;
}
/* Destroys an mxDictionary. Any contents are also destroyed. */
void mxDestroyDictionary(mxDictionary * victim) {
   mxObject * this;
   if (victim) {
      if (pthread_mutex_lock(victim->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", victim->name);
      } else {
         while (this = victim->start) {
            victim->start = victim->start->next; mxDestroyObject(this);
         }
         pthread_mutex_destroy(victim->gate); mxFree(victim->gate);
         mxFree(victim->name); mxFree(victim);
      }
   } else {
      mxPrintf("RCF: destroy a null dictionary.\n");
   }
}
/* Adds an object to an mxDictionary */                /* FIXME: FROM HERE */
void mxAddObject(mxDictionary * those, mxObject * this) {
   mxObject * o;
   if (those) {
      if (pthread_mutex_lock(those->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", those->name);
      } else {
         if (those->start) {
            those->start->previous = this; this->next = those->start;
            those->start = this; those->count++;
         } else {
```

```
            those->start = this; this->next = this->previous = NULL;
            those->count = 1;
         }
         pthread_mutex_unlock(those->gate);
      }
   } else {
      mxPrintf("RCF: adding an entry to a null dictionary.\n");
   }
}
/* Remove an object from an mxDictionary */
void mxRemoveObject(mxDictionary * those, mxObject * this) {
   if (those) {
      if (pthread_mutex_lock(those->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", those->name);
      } else {
         if (this->next) this->next->previous = this->previous;
         if (this->previous) this->previous->next = this->next;
            else those->start = this->next;
         this->next = this->previous = NULL;
         those->count--;
         pthread_mutex_unlock(those->gate);
      }
   } else {
      mxPrintf("RCF: removing an entry from a null dictionary.\n");
   }
}
/* Return first object with that root ID in that dictionary */
mxObject * mxGetObjectByRoot(mxDictionary * those, double r) {
   mxObject * this;
   mxObject * result = NULL;
   if (those) {
      if (pthread_mutex_lock(those->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", those->name);
      } else {
         this = those->start;
         while (this) {
            if (this->rootID == r) {
               #ifdef _MPX_DEBUG
               mxPrintf("[x] Found object with root %g.\n", r);
               #endif
               result = this; break;
            } else {
               this = this->next;
            }
         }
         pthread_mutex_unlock(those->gate);
      }
   } else {
      mxPrintf("RCF: searching a null dictionary.\n");
   }
   return result;
}
/* Return first object with that parent in that dictionary */
mxObject * mxGetObjectByParent(mxDictionary * those, double p) {
```

```
   mxObject * this;
   mxObject * result = NULL;
   if (those) {
      if (pthread_mutex_lock(those->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", those->name);
      } else {
         this = those->start;
         while (this) {
            if (this->rootParent == p) {
               result = this; break;
            } else {
               this = this->next;
            }
         }
         pthread_mutex_unlock(those->gate);
      }
   } else {
      mxPrintf("RCF: searching a null dictionary.\n");
   }
   return result;
}
/* Return first object with that local handle in that dictionary */
mxObject * mxGetObjectByHandle(mxDictionary * those, double h) {
   mxObject * this;
   mxObject * result = NULL;
   if (those) {
      if (pthread_mutex_lock(those->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", those->name);
      } else {
         this = those->start;
         while (this) {
            if (this->handle == h) {
               result = this; break;
            } else {
               this = this->next;
            }
         }
         pthread_mutex_unlock(those->gate);
      }
   } else {
      mxPrintf("RCF: searching a null dictionary.\n");
   }
   return result;
}
/* List a dictionary - diagnostic function */
void mxListDictionary(mxDictionary * those, int here) {
   int i;
   mxObject * c;
   if (those) {
      if (pthread_mutex_lock(those->gate)) {
         mxPrintf("RCF: lock failure on dictionary %s.\n", those->name);
      } else {
         mxPrintf("\n=== %s on node %d ===\n", those->name, here);
         c = those->start;
```

```
         while (c) {
            mxPrintf("%.6f, child of %.6f, locally %.6f\n",
                     c->rootID, c->rootParent, c->handle);
            c = c->next;
         }
         pthread_mutex_unlock(those->gate);
      }
   } else {
      mxPrintf("RCF: listing a null dictionary.\n");
   }
}
/* Take a vector of root handles and return a vector of local handles */
mxArray * mxGetLocalHandles(const mxArray * handles, mxDictionary * those) {
   if (those) {
      if (handles) {
         mxArray * newh = mxDuplicateArray(handles);        /* End result */
         double * local = mxGetPr(newh);        /* Adjusted handles */
         double * remote = mxGetPr(handles);        /* Original handles */
         int numh = mxGetNumberOfElements(handles);        /* How many */
         int i;                                          /* Jobbing counter */
         for (i=0; i<numh; i++) {
            if (remote[i] == 0.0) {
               local[i] = 0.0;
            } else {
               mxObject * this = mxGetObjectByRoot(those,remote[i]);
               if (this) {
                  local[i] = this->handle;
               } else {
                  mxPrintf("Invalid lookup: %d.\n", remote[i]);
                  local[i] = INVALID_HANDLE;
               }
            }
         }
         return newh;
      } else {
         return NULL;
      }
   } else {
      mxPrintf("RCF: Adjusting using a null dictionary?\n");
      return mxDuplicateArray(handles);
   }
}
/* Take a vector of local handles and return a vector of root handles */
mxArray * mxGetRootHandles(const mxArray * handles, mxDictionary * those) {
   if (those) {
      if (handles) {
         mxArray * newh = mxDuplicateArray(handles);        /* End result */
         double * local = mxGetPr(handles);        /* Original handles */
         double * remote = mxGetPr(newh);        /* Adjusted handles */
         int numh = mxGetNumberOfElements(handles);        /* How many */
         int i;                                          /* Jobbing counter */
         for (i=0; i<numh; i++) {
            if (local[i] == 0.0) {
               remote[i] = 0.0;
```

```
            } else {
               mxObject * this = mxGetObjectByHandle(those,local[i]);
               if (this) {
                  remote[i] = this->rootID;
               } else {
                  mxPrintf("RCF: invalid lookup %d.\n", local[i]);
                  remote[i] = INVALID_HANDLE;
               }
            }
         }
         return newh;
      } else {
         return NULL;
      }
   } else {
      mxPrintf("RCF: Adjusting using a null dictionary?\n");
      return mxDuplicateArray(handles);
   }
}
/* See if the first of a vector of handles exists in a dictionary */
int mxExists(const mxArray * handles, mxDictionary * those) {
   mxObject * this = NULL;
   double h;
   int result = FALSE;
   if (handles) {
      h = mxGetScalar(handles);
      if (those) {
         this = mxGetObjectByRoot(those, h);
         if (this) result = TRUE;
      }
   }
   return result;
}

/* The names of the callbacks */
const char * cnames[mxMAXCBACKS] = {
   "invalid", "Callback", "ButtonUpFcn", "ButtonDwnFcn",
   "MoveFcn", "DeleteFcn", "ResizeFcn", "ChangeFcn"
} ;

/* Updates an mxDictionary in light of a successful mxRequest.
   If an mxRequest has been successfully evaluated using engRequest,
   or has been received from a MEX-file (where presumably it was
   evaluated by the creating function), this function should be called
   exactly once. The third parameter points to a Matlab array containing
   a vector of handles. This array should have been created with
   mxGetLocalHandles; passing it here is merely an attempt to avoid a
   redundant dictionary lookup. */
void mxUpdateDictionary(mxDictionary * those, mxRequest * r, mxArray * h) {
   int nl, i, j;                /* Number of objects, counters */
   double * rlist;               /* List of remote handles affected */
   double * llist;                /* List of local equivalents */
   mxObject * this;               /* Jobbing graphics object */
   mxArray * that;               /* Jobbing mxArray */
```

```
if (r == NULL) return;        /* Reality check */
#ifdef _MPX_DEBUG
mxPrintf("[%d] Updating dictionary.\n", r->location);
#endif
if (h) {
   llist = mxGetPr(h); /* Get local handle list */
   #ifdef _MPX_DEBUG
   mxPrintf("[%d] Local handles start with %g.\n", r->location,llist[0]) ;
   #endif
}
if (r->object) {
   rlist = mxGetPr(r->object); /* Get remote handle list */
   #ifdef _MPX_DEBUG
   mxPrintf("[%d] Remote handles start with %g.\n", r->location,rlist[0]);
   #endif
}
switch(r->kind) {
   case NO_REQUEST: break;                        /* Er...nothing. */
   case SET_REQUEST:                             /* Update callbacks if any */
      for (i=0; i<mxMAXCBACKS; i++) {
         if (that = mxGetField(r->array, 0, cnames[i])) {
            for (j=0; j<nl; j++) {
               if (this = mxGetObjectByRoot(those, rlist[j])) {
                  char * cb = mxArrayToString(that);
                  mxSetObjectCallback(this, i, cb); mxFree(cb);
               }
            }
         }
      }
      break;
   case GET_REQUEST:        break;                          /* No response needed */
   case CREATE_REQUEST:                           /* Add new objects */
      if (nl = mxGetNumberOfElements(h)) {        /* If objects created... */
         for (i=0; i<nl; i++) {        /* ...add each to the records */
            this = mxCreateObject(llist[i], rlist[i], r->parent);
            for (j=0; j<mxMAXCBACKS; j++) {        /* and callbacks */
               if (that = mxGetField(r->array, 0, cnames[j])) {
                  char * cb = mxArrayToString(that);
                  mxSetObjectCallback(this, j, cb); mxFree(cb);
               }
            }
            mxAddObject(those, this);
            #ifdef _MPX_DEBUG
            mxPrintf("[%d] Adding %g to %s.\n", r->location, this->handle,
                     those->name);
            #endif
         }
      }
      break;
   case DESTROY_REQUEST:                           /* Remove the corpses */
      if (nl = mxGetNumberOfElements(h)) {        /* If objects destroyed */
         for (i=0; i<nl; i++) {
            if (this = mxGetObjectByRoot(those, rlist[i])) {
```

```
                int isFigure = (this->rootParent == 0.0);
                mxRemoveObject(those, this); mxDestroyObject(this);
                /* Remove children, and any grandchildren */
                while (this = mxGetObjectByParent(those, rlist[i])) {
                   if (isFigure) {        /* Grandchildren present; remove */
                      mxObject * kid;
                      while (kid=mxGetObjectByParent(those,this->rootID)) {
                         mxRemoveObject(those, kid); mxDestroyObject(kid);
                      }
                   }
                   mxRemoveObject(those, this); mxDestroyObject(this);
                }
             }
          }
       }
       break;
    case CALLBACK_REQUEST: break;               /* No response needed */
    case EVALUATE_REQUEST: break;               /* No response needed */
    default: ;                                 /* Silently ignore others */
  }
}

/* Adjusts a structure array of properties for this node's context.
   The first parameter is a structure array of properties which was created
   in the root context. The return value is an adjusted duplicate.
   Values which are graphics handles are replaced with local handles.
   Callback strings are adjusted to the appropriate stubs.
   The second parameter is the dictionary used for object lookups. */
mxArray * mxCensorProperties(const mxArray * old, mxDictionary * those) {
   mxArray * value;               /* An old value */
   char command[mxMAXCMD];        /* A new callback */
   mxArray * censored;               /* A new value */
   int i;                        /* Jobbing counter */
   mxArray * result;               /* The end result */

   if (result = mxDuplicateArray(old)) {
      /* Check through callbacks */
      for (i=0; i<mxMAXCBACKS; i++) {
         if (value = mxGetField(result, 0, cnames[i])) {
            if (mxIsEmpty(value)) continue;
            if (i == CALL_DELETE) {
               sprintf(command, "Callback(%d,gco); Closereq;", i);
            } else {
               sprintf(command, "Callback(%d,gco);", i);
            }
            censored = mxCreateString(command);        /* Create new value */
            mxSetField(result, 0, cnames[i], censored); /* Swap it in */
            mxDestroyArray(value);        /* Destroy old value */
         }
      }
      /* Check through handles */
      if (value = mxGetField(result, 0, "parent")) {
         mxSetField(result, 0, "parent", mxGetLocalHandles(value, those));
         mxDestroyArray(value);        /* Swap in local handles for parent */
```

```
      }
      if (value = mxGetField(result, 0, "children")) {
         mxSetField(result, 0, "children", mxGetLocalHandles(value, those));
         mxDestroyArray(value);        /* Swap in local handles for children */
      }
      /* FIXME: what of varying case ? */
   } else {
      mxPrintf("RCF: Attempting to censor a null array.\n");
   }
   return result;
}

#ifdef MATLAB_MEX_FILE

/* MEX-file argument parsers:
   Each of these takes pointers to the arguments, a current argument cursor,
   and the number of properties so far set. If the arguments under the
   cursor support it, they add property names to the names[] list, and the
   property values in the corresponding places in the values[] list, then
   update the cursor and number of properties and return TRUE. If not, they
   return FALSE. The result can be fed into the mxMakeStructure function
   to create an array field for an mxRequest. */

/* One generic property/value pair */
int mexParseOne(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 1) < nrhs) {        /* If sufficient arguments left... */
      if (mxIsChar(prhs[*cursor])) { /* ...and the next one is a property */
         names[*nfields] = mxArrayToString(prhs[*cursor]); (*cursor)++;
         values[*nfields] = mxDuplicateArray(prhs[*cursor]); (*cursor)++;
         (*nfields)++; return TRUE;
      }
   }
   return FALSE;
}
/* One color data matrix */
int mexParseC(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor) < nrhs) { /* If sufficient arguments left... */
      if (mxIsNumeric(prhs[*cursor])){
         names[*nfields]="CData";
         values[*nfields]=mxDuplicateArray(prhs[*cursor]);
         (*cursor)++; (*nfields)++; return TRUE;
      }
   }
   return FALSE;
}
/* Coordinates in two dimensions */
int mexParseXY(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 1) < nrhs) { /* If sufficient arguments left... */
       if (mxIsNumeric(prhs[*cursor])               /* ...all numbers */
          && mxIsNumeric(prhs[(*cursor)+1]) ){
             names[*nfields]="XData";
```

```
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            names[*nfields]="YData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            return TRUE;
         }
   }
   return FALSE;
}
/* Coordinates in two dimensions and color data */
int mexParseXYC(int nrhs, const mxArray * prhs[], int *cursor,
               int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 2) < nrhs) { /* If sufficient arguments left... */
       if (mxIsNumeric(prhs[*cursor])               /* ...all numbers */
          && mxIsNumeric(prhs[(*cursor)+1])
          && mxIsNumeric(prhs[(*cursor)+2]) ){
            names[*nfields]="XData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            names[*nfields]="YData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            names[*nfields]="CData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            return TRUE;
         }
   }
   return FALSE;
}
/* Coordinates in three dimensions */
int mexParseXYZ(int nrhs, const mxArray * prhs[], int *cursor,
               int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 2) < nrhs) { /* If sufficient arguments left... */
       if (mxIsNumeric(prhs[*cursor])               /* ...all numbers */
          && mxIsNumeric(prhs[(*cursor)+1])
          && mxIsNumeric(prhs[(*cursor)+2]) ){
            names[*nfields]="XData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            names[*nfields]="YData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            names[*nfields]="ZData";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor)++; (*nfields)++;
            return TRUE;
         }
   }
   return FALSE;
}
/* Coordinates in three dimensions, plus color data */
int mexParseXYZC(int nrhs, const mxArray * prhs[], int *cursor,
```

```
                int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 3) < nrhs) { /* If sufficient arguments left... */
       if (mxIsNumeric(prhs[*cursor])               /* ...all numbers */
          && mxIsNumeric(prhs[(*cursor)+1])
          && mxIsNumeric(prhs[(*cursor)+2])
          && mxIsNumeric(prhs[(*cursor)+3]) ){
             names[*nfields]="XData";
             values[*nfields]=mxDuplicateArray(prhs[*cursor]);
             (*cursor)++; (*nfields)++;
             names[*nfields]="YData";
             values[*nfields]=mxDuplicateArray(prhs[*cursor]);
             (*cursor)++; (*nfields)++;
             names[*nfields]="ZData";
             values[*nfields]=mxDuplicateArray(prhs[*cursor]);
             (*cursor)++; (*nfields)++;
             names[*nfields]="CData";
             values[*nfields]=mxDuplicateArray(prhs[*cursor]);
             (*cursor)++; (*nfields)++;
             return TRUE;
         }
   }
   return FALSE;
}
/* A point in two dimensions and a string to place there */
int mexParseXYT(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 2) < nrhs) { /* If sufficient arguments left... */
       if (mxIsNumeric(prhs[*cursor])                /* ...a number */
          && mxIsNumeric(prhs[(*cursor)+1])         /* ...a number */
          && mxIsChar(prhs[(*cursor)+2]) ){        /* ...a string */
             mxArray * where = mxCreateDoubleMatrix(1,3,mxREAL);
             double * coords = mxGetPr(where);
             names[*nfields]="Position"; values[*nfields]=where;
             coords[0] = mxGetScalar(prhs[*cursor]); (*cursor)++;
             coords[1] = mxGetScalar(prhs[*cursor]); (*cursor)++;
             coords[2] = 0.0; (*nfields)++; /* Z coordinate implied */
             names[*nfields]="String";
             values[*nfields]=mxDuplicateArray(prhs[*cursor]);
             (*cursor++); (*nfields)++;
             return TRUE;
         }
   }
   return FALSE;
}
/* A point in three dimensions and a string to place there */
int mexParseXYZT(int nrhs, const mxArray * prhs[], int *cursor,
                int *nfields, mxArray * values[], const char * names[]) {
   if ((*cursor + 3) < nrhs) { /* If sufficient arguments left... */
       if (mxIsNumeric(prhs[*cursor])                /* ...a number */
          && mxIsNumeric(prhs[(*cursor)+1])         /* ...a number */
          && mxIsNumeric(prhs[(*cursor)+2])         /* ...a number */
          && mxIsChar(prhs[(*cursor)+3]) ){        /* ...a string */
             mxArray * where = mxCreateDoubleMatrix(1,3,mxREAL);
             double * coords = mxGetPr(where);
```

```
            names[*nfields]="Position"; values[*nfields]=where;
            coords[0] = mxGetScalar(prhs[*cursor]); (*cursor)++;
            coords[1] = mxGetScalar(prhs[*cursor]); (*cursor)++;
            coords[2] = mxGetScalar(prhs[*cursor]); (*cursor)++; (*nfields)++;
            names[*nfields]="String";
            values[*nfields]=mxDuplicateArray(prhs[*cursor]);
            (*cursor++); (*nfields)++;
            return TRUE;
         }
   }
   return FALSE;
}

#else

/* Requests something of a Matlab engine.
   This function parses the mxRequest pointed to by the second argument, and
   issues commands to the Matlab engine pointed to by the first argument.
   The third argument is a dictionary of objects in this engine.
   Calls mxUpdateDictionary to record the update as appropriate.
   Returns a copy of the output from the commands.  */
char * engRequest(Engine * e, mxRequest * this, mxDictionary * those) {
   mxObject * that;                /* Object being pitilessly dissected */
   mxArray * local = NULL;        /* Local addresses for object(s) */
   mxArray * foo, * bar, *baz;        /* Perhaps a jobbing mxArray */
   char output[mxMAXOUT];        /* Perhaps an output buffer */
   char command[mxMAXCMD];        /* Perhaps a command */
   char * result = NULL;        /* Final output from engine */

   /* Zero an output buffer and assign it to this engine */
   memset(output, 0, mxMAXOUT); engOutputBuffer(e, output, mxMAXOUT);
   /* Examine the request */
   switch(this->kind) {
      case NO_REQUEST:        /* Null request - reserved for future use */
         mxPrintf("Something prodded me.\n"); break;
      case SET_REQUEST:        /* Set the properties of local objects */
         /* Get local handles and put them in the workspace */
         local = mxGetLocalHandles(this->object, those);
         mxSetName(local, "foo_"); engPutArray(e, local);
         /* Set properties using a structure array of values */
         bar = mxCensorProperties(this->array, those);
         mxSetName(bar, "bar_"); engPutArray(e, bar); mxDestroyArray(bar);
         engEvalString(e, "set(foo_,bar_); clear foo_ bar_;");
         break;
      case GET_REQUEST:        /* Get a property of a local object */
         /* Do nothing at all because it's been handled by the root */
         mxPrintf("Inspecting, O Master!\n"); break;
      case CREATE_REQUEST:        /* Create a copy of an object */
         bar = mxCreateScalar(this->parent);        /* Get local parent handle */
         foo = mxGetLocalHandles(bar, those); mxDestroyArray(bar);
         mxSetName(foo, "foo_"); engPutArray(e, foo);        /* Send to engine */
         sprintf(command, "bar_ = %s('Parent',foo_);", this->words);
         #ifdef _MPX_DEBUG
         mxPrintf("[%d] Command: %s\n", this->location,command);
```

```
      #endif
      engEvalString(e, command); bar = engGetArray(e, "bar_"); /* Create */
      if (mxIsEmpty(bar)) {        /* Failure */
         mxPrintf("[%d] RCF: Creation of %s failed\n",
                  this->location, this->words);
      } else {        /* Success; initialize */
         if (this->array) {
            baz = mxCensorProperties(this->array, those);
            mxSetName(baz, "baz_"); engPutArray(e, baz);
            engEvalString(e, "set(bar_,baz_); clear bar_ baz_;");
            mxDestroyArray(baz);
         }
         local = bar;        /* Save local handles for records */
      }
      break;
   case DESTROY_REQUEST:         /* Destroy something that isn't the root */
      /* Get local handles, delete local objects */
      local = mxGetLocalHandles(this->object, those);
      mxSetName(local, "foo_"); engPutArray(e, local);
      engEvalString(e, "delete(foo_); clear foo_;");
      break;
   case CALLBACK_REQUEST:        /* Evaluate a callback */
      /* Update GUI component's value */
      if (that = mxGetObjectByRoot(those, mxGetScalar(this->object))) {
         foo = mxCreateScalar(that->handle); mxSetName(foo, "foo_");
      } else {
         mxPrintf("[%d] RCF: There is no %g.\n", this->location,
                  mxGetScalar(this->object));
         break;
      }
      bar = mxDuplicateArray(this->array); mxSetName(bar, "bar_");
      engPutArray(e, foo); engPutArray(e,bar);
      engEvalString(e, "set(foo_, 'Value', bar_); clear foo_ bar_;");
      /* Execute callback if at the root */
      if (this->location == ROOT) {
         strcpy(command, mxGetObjectCallback(that, this->status));
         #ifdef _MPX_DEBUG
         mxPrintf("[%d] Executing [%s].\n", this->location, command);
         #endif
         engEvalString(e, command);
      }
      break;
   case EVALUATE_REQUEST:        /* Execute an arbitrary Matlab expression */
      engEvalString(e, this->words); break;
   default:                      /* The request is in error */
      mxPrintf("RCF: A request of type %d?\n", this->kind);
  }
  if (local) mxUpdateDictionary(those, this, local);        /* Update records */
  if (local) mxDestroyArray(local);                /* Destroy local handles */
  return strdup(output);        /* Return output from Matlab engine */
}

#endif
```

A.2 The `legion` daemon

legion.c

```
/*
** file: legion.c
**
** purpose: implementation for the Concurrent Matlab legion daemon
**
** created: 3/2000 by AHD
*/

#include "mpx.h"

/* My current rank, nodes in current communicator */
int me, nodes;

/* Standing queues of requests */
mxQueue * outQ;                /* Outbound messages */
mxQueue * inQ;        /* Inbound messages */

/* Standing dictionary of graphics objects */
mxDictionary * objects;

/* Mutex to protect stdio */
pthread_mutex_t stdioMutex;

/* And its' constructor and destructor */
void stdInit(void) { pthread_mutex_init(&stdioMutex, NULL); }
void stdFinalize(void) { pthread_mutex_destroy(&stdioMutex); }

/* Threads spawned by main() */
pthread_t keythread, engthread, tapthread;

/* Termination flag - synchronized between threads */
int continuing;
pthread_mutex_t contMutex;

/* Set up continuing flag - not for call by a thread */
int contInit(void) {
   pthread_mutex_init(&contMutex, NULL); continuing = TRUE;
}
/* Set continuing flag */
void contSet(int newValue) {
   if (pthread_mutex_lock(&contMutex)) {
      mxPrintf("Reality check failed - cannot set continuing flag.\n");
   } else {
      continuing = newValue; pthread_mutex_unlock(&contMutex);
   }
}
/* Query continuing flag */
int contQuery(void) {
   int result = FALSE;
```

```
   if (pthread_mutex_lock(&contMutex)) {
      mxPrintf("Reality check failed - cannot query continuing flag.\n");
   } else {
      result = continuing; pthread_mutex_unlock(&contMutex);
   }
   return result;
}
/* Take down continuing flag - not for call by a thread */
void contFinalize(void) {
   if (pthread_mutex_destroy(&contMutex)) {
      mxPrintf("Reality check failed - continuing mutex unkillable.\n");
   }
}


/* Thread to consume entries in the local queue (output to stdout) */
void * engine(void * eng) {
   int stillGoing = TRUE;        /* Local copy of continuing */
   mxRequest * this;                  /* The current request */
   char * output;                /* Its' result */
   mxObject *that, *thother;        /* Object(s) under scrutiny */
   mxArray * answer;                /* Result of a Matlab expression */

   while (stillGoing) {                        /* While process still alive... */
      while (this = mxDequeue(inQ)) { /* While requests queued ... */
         if (mxIsMagicBullet(this)) {
            contSet(FALSE); /* Signal other threads to commit suicide */
            stillGoing = FALSE;        /* Likewise here */
         } else {
            output = engRequest(eng, this, objects);        /* Answer request */
            if (this->kind == EVALUATE_REQUEST)  {
               /* Print Matlab's response, skipping the prompt if present */
               pthread_mutex_lock(&stdioMutex);
               if (strncmp(output,">>",2)) mxPrintf("%s", output);
                  else mxPrintf("%s", output+2);
               pthread_mutex_unlock(&stdioMutex);
               /* If this is a request from the keyboard, disappear it */
               if (mxIsLocal(this)) {
                  mxDestroyRequest(this); this = NULL;
               }
            }
            mxFree(output);         /* Dispose of output buffer */
         }
         if (this) mxEnqueue(this, outQ);        /* Pass on to next node */
         #ifdef _MPX_DEBUG
         mxListDictionary(objects, me);
         #endif
      }
   }
   return NULL;
}

/* Thread to handle requests from the local pipe */
void * tap(void * path) {
```

```
int state = WORKING;
int querent, dummy;
mxRequest * this;
mxObject * that;
mxArray * handles;

/* Create and open FIFO, supplying name if necessary */
if (path == NULL) path = "/tmp/legion.pipe";
mkfifo(path, S_IRUSR|S_IWUSR|S_IXUSR);
querent = open(path, O_RDONLY);
if (querent > 0) {
   state = WORKING;
} else {
   state = STOPPED;
}

/* Read requests from the FIFO and process them */
/* Busy wait could be replaced with SIGIO handling, but not yet */
while (state != STOPPED) {
   if (this = mxReadRequest(querent, me)) {
      this->origin = me;        /* New request, originated from here */
      switch(this->kind) {
         case NO_REQUEST:
            mxEnqueue(this, outQ); break;
         case SET_REQUEST:
            handles = mxGetLocalHandles(this->object, objects);
            mxUpdateDictionary(objects, this, handles);
            mxDestroyArray(handles); mxEnqueue(this, outQ); break;
         case GET_REQUEST:
            mxEnqueue(this, outQ); break;
         case CREATE_REQUEST:
            if (mxExists(this->object, objects)) {
               mxDestroyRequest(this);
               #ifdef _MPX_DEBUG
               mxPrintf("[%d] already has %s %g.\n", me, this->words,
                        mxGetScalar(this->object));
               #endif
            } else {
               handles = mxDuplicateArray(this->object);
               mxUpdateDictionary(objects, this, handles);
               mxDestroyArray(handles); mxEnqueue(this, outQ);
            }
            break;
         case DESTROY_REQUEST:
            handles = mxGetLocalHandles(this->object, objects);
            mxUpdateDictionary(objects, this, handles);
            mxDestroyArray(handles); mxEnqueue(this, outQ); break;
         case CALLBACK_REQUEST:         /* Callback */
            #ifdef _MPX_DEBUG
            mxPrintf("[%d] Callback %d of object %g activated.\n",
              this->location, this->status, mxGetScalar(this->object));
            #endif
            if (me != ROOT) {
               /* Normalize handles and send around the ring */
```

```
                handles = mxGetRootHandles(this->object, objects);
                mxDestroyArray(this->object); this->object = handles;
                mxEnqueue(this, outQ);
             } else {
                /* Run it past the local engine, first */
                mxEnqueue(this, inQ);
             }
             break;
          case EVALUATE_REQUEST:
             mxEnqueue(this, outQ); break;
          default:
              mxPrintf("RCF - request for %d.\n", this->kind);
              mxDestroyRequest(this); this = NULL;
       }
    }
    if (contQuery() == TRUE) {
       state = WORKING;
       pthread_yield();
    } else {
       state = STOPPED;
    }
 }

 /* Clean up */
 close(querent);
 remove(path);
 return NULL;
}

/* TRUE if all the characters in the string are whitespace */
int isempty(const char * guinea) {
 char yes = TRUE;
 while (*guinea && yes) yes = isspace(*(guinea++)) && yes;
 return yes;
}

/* TRUE if the first four characters in the string are "quit" */
int isquit(const char * guinea) {
 return (strncmp(guinea, "quit",4)==0);
}

/* Thread to feed input from a stream into the local queue */
void * keyboard(void * stream) {
 char input[mxMAXCMD];        /* Input buffer */
 char * prompt = "[?] ";        /* Prompt string */
 mxRequest * this;                /* Current request */
 int stillGoing = TRUE;        /* Local termination flag */

 while (stillGoing) {
    pthread_mutex_lock(&stdioMutex);
    fprintf(stdout, prompt); fgets(input, mxMAXCMD, stream);
    if (isempty(input)) {
       /* Don't do anything - this is an empty command */
    } else if (isquit(input)) { /* Shut down */
```

```
         mxEnqueue(mxCreateMagicBullet(me),inQ);
         stillGoing = FALSE;
      } else { /* Matlab command - request evaluation */
         this = mxCreateRequest(EVALUATE_REQUEST, me, me);
         this->words = strdup(input);
         mxEnqueue(this, inQ);
      }
      pthread_mutex_unlock(&stdioMutex);
      pthread_yield();
   }
   return NULL;
}

/* Main thread; talks to other processes, administers other threads */
int main(int argc, char *argv[]) {
   Engine * e;                                  /* Matlab engine for this daemon */
   char * engineOpener = "/opt/bin/matlab";        /* command to get one */
   pthread_attr_t threadkind;                        /* their attributes */
   MPI_Comm comm=MPI_COMM_WORLD;        /* all-nodes intracommunicator */
   int left, right;                        /* next, previous neighbours */
   mxRequest * this;                        /* current message */
   int stillGoing = TRUE;                /* local termination flag */
   int state = WORKING;                        /* message passing state */

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(comm, &me); MPI_Comm_size(comm, &nodes);
   left = (me+1) % nodes; right = (me + nodes -1) % nodes;
   #ifdef _MPX_DEBUG
   mxPrintf("I am daemon %d, O Master. [%d,%d]\n", me, left, right);
   #endif
   if (e = engOpen(engineOpener)) {
      /* Initialize queues, mutexes, and threads */
      contInit();                           /* Arm termination flag */
      outQ = mxInitQueue("outgoing");        /* Set up outbound message queue */
      inQ = mxInitQueue("local");        /* Set up local message queue */
      stdInit();                          /* Arm standard I/O mutex */
      objects = mxCreateDictionary("objects");        /* Set up object dictionary */
      pthread_attr_init(&threadkind);        /* Threads created attached */
      pthread_attr_setdetachstate(&threadkind, PTHREAD_CREATE_UNDETACHED);
      pthread_create(&engthread, &threadkind, engine, e);
      pthread_create(&tapthread, &threadkind, tap, NULL);
      if (me == ROOT) pthread_create(&keythread, &threadkind, keyboard, stdin);
      /* Talk to other processes until a magic bullet hits you */
      while (state != STOPPED) {
         /* while there are messages from the right, receive them */
         if (this = mxRecvRequest(right, MPI_COMM_WORLD, me)) {
            if (state == WORKING) {
               if (mxIsLocal(this)) {
                  mxDestroyRequest(this);
               } else {
                  mxEnqueue(this, inQ);
               }
            } else if (state == LISTENING) {
               if (mxIsMagicBullet(this)) {
```

```
               #ifdef _MPX_DEBUG
               mxPrintf("[%d] They got me, boss!\n", this->location);
               #endif
               state = STOPPED;
            } else {
               mxDestroyRequest(this);
            }
         }
      }
      /* while there are outgoing messages, send them to the left */
      if ((state == WORKING) && (this = mxDequeue(outQ))) {
         if (mxIsMagicBullet(this)) {
            if (mxIsLocal(this)) {
               state = LISTENING;
            } else {
               state = STOPPED;
            }
            #ifdef _MPX_DEBUG
            mxPrintf("[%d] Sending bullet.\n", this->location);
            #endif
            mxSendRequest(left, comm, this, me);
         } else {
            #ifdef _MPX_DEBUG
            mxPrintf("[%d] Sending packet.\n", this->location);
            #endif
            mxSendRequest(left, comm, this, me);
         }
      }
      pthread_yield();
   }
   /* Wait for other threads to terminate */
   if (me == ROOT) {
      pthread_join(keythread, NULL);
      #ifdef _MPX_DEBUG
      mxPrintf("[%d] Keyboard down.\n", me);
      #endif
   }
   pthread_join(engthread, NULL);
   #ifdef _MPX_DEBUG
   mxPrintf("[%d] Engine down.\n", me);
   #endif
   #ifdef THIS_SECTION_DYKED_OUT
   pthread_join(tapthread, NULL);
   mxPrintf("[%d] Pipe down.\n", me);
   #endif
   /* Tidy up */
   mxDestroyDictionary(objects);        /* Clean up object dictionary */
   mxFinalizeQueue(outQ);         /* Shut down outbound message queue */
   mxFinalizeQueue(inQ);        /* Shut down local request queue */
   stdFinalize();                /* Disarm the stdio mutex */
   contFinalize();                /* Disarm global termination flag */
   engClose(e);                /* Lose the Matlab engine */
   MPI_Finalize();                /* Shut down the message passing */
} else {
```

```
      MPI_Abort(MPI_COMM_WORLD); /* Infrastructure failure: messy abort! */
   }
}
```

A.3 The stubs

These listings compile into MEX-files for direct use in manipulating glyphs. Their functions are described in section 3.2.3.

Constructors

Below are a representative sampling of the constructor methods used in Concurrent Matlab over MPI. There are as many again not shown, which are of exactly the same pattern.

Figure window

```
/* MEX-file to implement the Figure function */

#include "mpx.h"


/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   char * path = "/tmp/legion.pipe";        /* Its' name */
   int result;                /* Jobbing status variable */
   mxArray * handle[1];        /* Figure being created */
   double h;                /* Its' handle */
   int cursor;                /* Current argument being parsed */
   int n = 0;                /* Current number of initial properties */
   const char * names[256];        /* Property names */
   mxArray * values[256];        /* Property values */
   mxArray * properties = NULL;                /* A structure containing them */
   mxRequest * missive = NULL;                /* Message to the daemon */

   mexSetTrapFlag(TRUE);        /* Errors trap here */

   /* Call constructor and get figure handle */
   result = mexCallMATLAB(1, handle, nrhs, (mxArray **)prhs, "figure");

   if (result) {
      printf("Cannot create this figure.\n");
   } else {
```

```
      if (nlhs) plhs[0] = handle[0];        /* Pass back handle to caller */

      /* Send a creation request for this figure */
      pipeline = open(path, O_WRONLY);        /* Blocking call */
      if (pipeline > 0) {
         if (nrhs > 2) {        /* Properties need setting, too */
            cursor = nrhs % 2;        /* Skip handle if first argument */
            do {
               result = mexParseOne(nrhs, prhs, &cursor, &n, values, names);
            } while (result);
            if (n) properties = mxMakeStructure(values, names, n);
            if (properties) {
               #ifdef _MPX_DEBUG
               mxPrintf("Properties! Yay! \n");
               #endif
            }
         }
         missive = mxCreateRequest(CREATE_REQUEST, ROOT, ROOT);
         missive->object = mxDuplicateArray(handle[0]);
         missive->parent = 0.0;
         missive->words = "figure"; missive->array = properties;
         mxWriteRequest(missive, pipeline);
         /* Rely on MATLAB to clean up dynamically allocated memory */
         close(pipeline);
      } else {
         mexPrintf("Reality check failed: cannot open %s.\n", path);
      }
   }
}
```

Set of plot axes

```
/* MEX-file to implement the Axes function */

#include "mpx.h"

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   char * path = "/tmp/legion.pipe";        /* Its' name */
   mxArray * answer[1];        /* Jobbing answer */
   int result;                /* Jobbing status variable */
   mxArray *h, *p;        /* Handles of child and parent */
   mxRequest * prelude;        /* Creation request for parent figure */
   mxRequest * missive;        /* Creation request for child axes */
   int cursor;                /* Current argument being parsed */
   int nfields;                /* Number of properties set to date */
   mxArray * values[256];        /* Property values */
   const char * names[256];        /* Property names */

   /* Call constructor */
```

```
   mexSetTrapFlag(TRUE);        /* Errors trap here */
   result = mexCallMATLAB(1, answer, nrhs, (mxArray **)prhs, "axes");

   if (result) {
      printf("Cannot create these axes.\n");
   } else {
      h = answer[0];                                /* Acquire axes handle */
      if (nlhs) plhs[0] = answer[0];        /* Pass back handle to caller */
      p = mxDuplicateArray(mexGet(mxGetScalar(h),"Parent"));/* Acquire parent */

      pipeline = open(path, O_WRONLY);        /* Open pipe for transmission */
      if (pipeline > 0) {
         /* Create parent figure, if necessary */
         prelude = mxCreateRequest(CREATE_REQUEST, 0, 0);
         prelude->parent = 0.0; prelude->words = "figure";
         prelude->object = p; mxWriteRequest(prelude, pipeline);
         /* Create axes */
         missive = mxCreateRequest(CREATE_REQUEST, 0, 0);
         missive->parent = mxGetScalar(p); missive->words = "axes";
         missive->object = mxDuplicateArray(h);
         if (nrhs > 2) {        /* Get additional properties if any */
            #ifdef _MPX_DEBUG
            mxPrintf("Properties! Yay!\n");
            #endif
            cursor = nrhs % 2;        /* Skip handle if it was first */
            nfields = 0;
            while (mexParseOne(nrhs,prhs,&cursor,&nfields,values,names) );
            missive->array = mxMakeStructure(values, names, nfields);
         }
         mxWriteRequest(missive, pipeline);
         if (nlhs > 1) plhs[1] = mxRequestToStructure(missive);
         close(pipeline);

      } else {
         mexPrintf("Reality check failed: cannot open %s.\n", path);
      }
   }
}
```

Line

```
/* MEX-file to implement the Line function */

#include "mpx.h"

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   char * path = "/tmp/legion.pipe";        /* Its' name */
   mxArray * answer[1];        /* Jobbing answer */
   int result;                /* Jobbing status variable */
```

```
   mxArray *h, *p, *gp;        /* Handles of grandparent, parent and child */
   mxRequest * prelude;        /* Creation request for parent figure */
   mxRequest * missive;        /* Creation request for child axes */
   int cursor;                /* Current argument being parsed */
   int nfields;                /* Number of properties set to date */
   mxArray * values[256];        /* Property values */
   const char * names[256];        /* Property names */

   /* Call constructor */
   mexSetTrapFlag(TRUE);        /* Errors trap here */
   result = mexCallMATLAB(1, answer, nrhs, (mxArray **)prhs, "line");

   if (result) {
      printf("Cannot create this line.\n");
   } else {
      h = answer[0];                                /* Acquire line handle */
      if (nlhs) plhs[0] = answer[0];        /* Pass back handle to caller */
      p = mxDuplicateArray(mexGet(mxGetScalar(h),"Parent")); /* Acquire axes */
      gp = mxDuplicateArray(mexGet(mxGetScalar(p),"Parent")); /* And figure */

      pipeline = open(path, O_WRONLY);        /* Open pipe for transmission */
      if (pipeline > 0) {
         /* Create grandparent figure, if necessary */
         prelude = mxCreateRequest(CREATE_REQUEST, 0, 0);
         prelude->parent = 0.0; prelude->words = "figure";
         prelude->object = gp; mxWriteRequest(prelude, pipeline);
         /* Create parent axes, if necessary */
         prelude->parent = mxGetScalar(gp); prelude->words = "axes";
         prelude->object = p; mxWriteRequest(prelude, pipeline);
         /* Create figure */
         missive = mxCreateRequest(CREATE_REQUEST, 0, 0);
         missive->parent = mxGetScalar(p); missive->words = "line";
         missive->object = h;
         /* Get additional properties if any */
         cursor = nfields = 0;        /* Start from first argument */
         if (mexParseXYZ(nrhs, prhs, &cursor, &nfields, values, names)) {
            /* Add three coordinate arrays */
         } else if (mexParseXY(nrhs, prhs, &cursor, &nfields, values, names)) {
            /* Add two coordinate arrays */
         }
         while (mexParseOne(nrhs,prhs,&cursor,&nfields,values,names)) ;
         if (nfields) missive->array = mxMakeStructure(values, names, nfields);
         /* Send request */
         mxWriteRequest(missive, pipeline);
         close(pipeline);
      } else {
         mexPrintf("Reality check failed: cannot open %s.\n", path);
      }
   }
}
```

User interface control

```
/* MEX-file to implement the Uicontrol function */

#include "mpx.h"

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   char * path = "/tmp/legion.pipe";        /* Its' name */
   mxArray * answer[1];        /* Jobbing answer */
   int result;                /* Jobbing status variable */
   mxArray *h, *p;        /* Handles of child and parent */
   mxRequest * prelude;        /* Creation request for parent figure */
   mxRequest * missive;        /* Creation request for child uicontrol */
   int cursor;                /* Current argument being parsed */
   int nfields;                /* Number of properties set to date */
   mxArray * values[256];        /* Property values */
   const char * names[256];        /* Property names */
   mxArray * callback;        /* Jobbing callback value */

   /* Call constructor */
   mexSetTrapFlag(TRUE);        /* Errors trap here */
   result = mexCallMATLAB(1, answer, nrhs, (mxArray **)prhs, "uicontrol");

   if (result) {
      printf("Cannot create this uicontrol.\n");
   } else {
      h = answer[0];                                /* Acquire uicontrol handle */
      if (nlhs) plhs[0] = answer[0];        /* Pass back handle to caller */
      p = mxDuplicateArray(mexGet(mxGetScalar(h),"Parent")); /* Get figure */

      /* Set callbacks correctly - fix so others are right, also*/
      if (mexGet(mxGetScalar(h),"Callback")) {
         callback = mxCreateString("Callback(1,0)");
         mexSet(mxGetScalar(h),"Callback",callback);
         mxDestroyArray(callback);
      }

      pipeline = open(path, O_WRONLY);        /* Open pipe for transmission */
      if (pipeline > 0) {
         /* Create parent figure, if necessary */
         prelude = mxCreateRequest(CREATE_REQUEST, 0, 0);
         prelude->parent = 0.0; prelude->words = "figure";
         prelude->object = p; mxWriteRequest(prelude, pipeline);
         /* Create uicontrol */
         missive = mxCreateRequest(CREATE_REQUEST, 0, 0);
         missive->parent = mxGetScalar(p); missive->words = "uicontrol";
         missive->object = h;
         if (nrhs > 2) {        /* Get additional properties if any */
            cursor = nrhs % 2;        /* Skip handle if it was first */
            nfields = 0;
            while (mexParseOne(nrhs,prhs,&cursor,&nfields,values,names)) ;
            missive->array = mxMakeStructure(values, names, nfields);
```

```
         }
         mxWriteRequest(missive, pipeline);
         close(pipeline);
      } else {
         mexPrintf("Reality check failed: cannot open %s.\n", path);
      }
   }
}
```

Other methods

Callbacks

```
/* MEX-file to implement the Callback function */

#include "mpx.h"

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   int result;               /* Jobbing status variable */
   mxRequest * missive;        /* Message to engine daemon */
   char * path = "/tmp/legion.pipe";

   if (nrhs) {        /* If any arguments */
      if (mxIsNumeric(prhs[0])) {  /* And the first one is a callback index */
         pipeline = open(path, O_WRONLY);
         /* Create a callback mxRequest with the appropriate callback */
         missive = mxCreateRequest(CALLBACK_REQUEST, ROOT, ROOT);
         missive->status = (int)(mxGetScalar(prhs[0]));        /* Which callback */
         /* Add the object handle, and its' current value */
         missive->object = mxDuplicateArray(mexGet(0.0, "CallbackObject"));
         missive->array = mxDuplicateArray(
            mexGet(mxGetScalar(missive->object), "Value"));
         mxWriteRequest(missive, pipeline);
         close(pipeline);
      }
   }
}
```

Destructor

```
/* MEX-file to implement the Delete function */

#include "mpx.h"
```

```
/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   int result;                /* Jobbing status variable */
   mxRequest * missive;        /* Message to engine daemon */
   char * path = "/tmp/legion.pipe";

   if (nrhs) {        /* If any arguments */
      mexSetTrapFlag(TRUE);  /* Call delete handler, trap errors to here */
      result = mexCallMATLAB(nlhs, plhs, nrhs, (mxArray **)prhs, "delete");
      if (result) {        /* Failure */
         mexPrintf("That didn't work.\n");
      } else {                /* Success - write delete request for first argument */
         if (mxIsNumeric(prhs[0])) {        /* ...if it is a handle */
            pipeline = open(path, O_WRONLY);
            missive = mxCreateRequest(DESTROY_REQUEST, ROOT, ROOT);
            missive->object = (mxArray *)prhs[0];
            mxWriteRequest(missive, pipeline);
            close(pipeline);
         }
      }
   }
}
```

Update

```
/* MEX-file to implement the Set function */

#include "mpx.h"

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   int result;                /* Jobbing status variable */
   int cursor;                /* Current update being sent */
   char * path = "/tmp/legion.pipe";        /* Pipe to write to */
   const char * names[256];        /* List of properties */
   mxArray * values[256];        /* List of values */
   int nfields;                        /* How many in those lists */
   mxArray * properties;        /* The above as a structure array */
   mxRequest * missive;                /* A request */


   mexSetTrapFlag(TRUE);        /* Errors trap to here */
   result = mexCallMATLAB(nlhs, plhs, nrhs, (mxArray **)prhs, "set");
   if (result) {
      mexPrintf("That didn't work.\n");
   } else {
      if (nrhs > 1) {
         /* Read property/value pairs from the argument list */
         cursor = 1; nfields = 0;        /* Skip first argument (a handle) */
```

```
         while (mexParseOne(nrhs, prhs, &cursor, &nfields, values, names)) ;
         if (nfields > 0) {        /* If there are properties to set, send them */
            if (pipeline = open(path, O_WRONLY)) {
               missive = mxCreateRequest(SET_REQUEST, ROOT, ROOT);
               missive->object = mxDuplicateArray(prhs[0]);
               properties = mxMakeStructure(values, names, nfields);
               missive->array = properties;
               mxWriteRequest(missive, pipeline);
               close(pipeline);
            } else {
               mxPrintf("Reality check failed: cannot open pipe %s\n", path);
            }
         }
         if (nlhs > 0) plhs[0] = properties;
      }
   }
}
```

Query

```
/* MEX-file to implement the Get function */

#include "mpx.h"

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
   int pipeline;        /* The pipe to the rest of the world */
   int result;               /* Jobbing status variable */
   double handle;        /* Object being queried */
   char * path = "/tmp/legion.pipe";

   /* Hand this off to the proper function */
   mexCallMATLAB(nlhs, plhs, nrhs, (mxArray **)prhs, "get");

   /* No point to these yet, maybe never */
   /* pipeline = open(path, O_WRONLY); */
   /* missive = mxCreateRequest(GET_REQUEST, ROOT, ROOT); */
   /* missive->handle = prhs[0]; missive->words = prhs[1]; */
   /* mxWriteRequest(missive, pipeline) */
   /* close(pipeline); */
}
```

Appendix B

Concurrent Matlab under RMI

B.1 Facet hierarchy

The Java interfaces in this section represent the basic roles a facet must play. The Java classes in this section are default implementations using the Remote Method Invocation infrastructure provided by Java. For more information, please see section 3.3, especially figure 3.4.

Interface `Peer`

```
package mirror;

import java.io.*;
import java.rmi.*;

// A mirror is a collection of peers, which use this interface to
// communicate. A peer adds or removes other peers from its' broadcast list
// as appropriate. A peer has a shared state, consisting of a set of metadata
// and a primary value. These are traded around as Facts. Peers present new
// Facts to each other using amend(). Peers evaluate and return their values
// on request - this may have side effects at the peer's tree, which is how
// callbacks come about. Peers will clone an inactive copy of their subtree
// on request. This may then be grafted into another tree.

public interface Peer extends Remote {
```

```
   boolean amend(Fact f) throws IOException;
   Object evaluate(Object key, int mode) throws IOException;
   Fact add(Name candidate, Name referee) throws IOException;
   boolean remove(Name candidate) throws IOException;
   boolean graft(MarshalledRune cutting) throws IOException;
   MarshalledRune toCutting() throws IOException;
}
```

Interface `Ancestor`

```
package mirror;

import java.io.*;

// An ancestor is a peer of the root mirror of a forest of facet trees. In
// addition to its' role with respect to other peers, it maintains the local
// listings of a distributed mirror registry, which it acquires from members
// of its' local tree. It also maintains a name supply. Names generated from
// this source are unique across the forest for the lifetime of the tree.

public interface Ancestor extends Peer {
   boolean bind(Name binding) throws IOException;
   boolean loose(Name binding) throws IOException;
   Name complete(Name binding) throws IOException;
   Object freshIdentifier(Object context) throws IOException;
}
```

Interface `Child`

```
package mirror;

import java.io.*;

// A child is that which may be adopted as a descendent of a node in a facet
// tree. Children know their parent, the root of their tree, the location of
// their tree, and possess a reference to an external implementation. These
// things are set at adoption time, and changed at subsequent adoption times,
// until disposal.

public interface Child {

   // The possible states.
   final int INACTIVE = 0;
   final int ACTIVE = 1;
   final int DEAD = 2;

   boolean adopt(Parent p, Ancestor a, int location, Object backing[]);
   int status();
   void dispose();
}
```

```
// There exists a corresponding Parent interface.
```

Interface `Parent`

```
package mirror;

// A parent is a node in a facet tree, which may adopt and disown children.
// Facet trees grow by grafting facets of a mirror onto a parent. If the
// mirror does not exist, it is created. If the mirror exists elsewhere, it
// is imported. If the mirror exists in this tree, then it is moved to this
// point. If there is a subtree rooted at the facet, it follows. This explains
// why it is important not to graft a mirror onto one of its' descendants.

public interface Parent {
   boolean disown(Child c);
   boolean adopt(Child c);
   Child graft(String mirror, Object details[]);
   int status();
   void dispose();
}
```

Class `mirror.Rune`

```
package mirror;

import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

// A Rune is a facet of a mirror which owns a shared value and metadata about
// that value. This class provides a default implementation of runes. Each
// facet of a mirror has a local backing instance. This backing instance is
// very implementation-dependent and must be defined by overriding methods
// in a subclass.

public class Rune extends UnicastRemoteObject implements Peer, Child, Parent {
   Vector children;                // the set of children
   transient Parent parent;        // the parent of this facet
   transient Ancestor ancestor;        // the root of this tree
   transient Hashtable peers;        // the set of known peers, hashed by node
   transient int location;        // the number of this tree
   String identity;                // the identity of this mirror

   Hashtable properties;        // local copy of the metadata
   Object primary;                // local copy of the primary value
   Object primaryKey;                // designator for primary value in metadata
   transient int state;                // state of this facet
   boolean isStructural;        // true if this facet is structural
```

```
transient Vector queue;        // queue of facts awaiting processing

// A daemon thread for doing odd jobs. Runs queued Runnables.
static Executor executor;

// A wildcard or unallocated location
public final static int NO_LOCATION = -1;

// Some sundry evaluation modes
final static int EVALUATE = 0;
final static int EXECUTE = 1;

// Do nothing callback
static Object NO_OP = "";

// Construct an inactive facet of this mirror from scratch.
// In inactive state, a rune has no tree, mirror or backing instance.
public Rune(String mirror) throws IOException {
   this.identity = mirror; this.location = NO_LOCATION;
   this.children = new Vector(); this.peers = new Hashtable();
   this.properties = new Hashtable();
   this.primary = null; this.primaryKey = "value";
   this.parent = null; this.ancestor = null;
   this.state = INACTIVE; this.isStructural = true;
   this.queue = new Vector();
}

// Deserialize an inactive facet of this mirror.
private void readObject(ObjectInputStream stream)
   throws IOException, ClassNotFoundException {
   stream.defaultReadObject();
   this.parent = null; this.ancestor = null;
   this.peers = new Hashtable(); this.location = NO_LOCATION;
   this.state = INACTIVE; this.isStructural = true;
   this.queue = new Vector();
}

// ***** A simple interface towards the outside world ***** //

// Set an item of this facet's data.
// Returns null on failure. Returns the previous value on success.
// Was synchronized, will be again one day.
public Object set(Object key, Object value) throws IOException {

   // Reality checks
   if (key == null) return null;
   if (state == DEAD) {
      log.println(identity + " cannot accept metadata operations.");
      return null;
   }

   // Amend the metadata
   Object old;
   Fact fact = (Fact)(properties.get(key));        // The fact in question
```

```
   if (fact == null) {        // ... doesn't yet exist
      fact = isExecutableKey(key)  ? new Executable(key, value, location)
                                   : new Fact(key, value, location);
      properties.put(fact,fact);
      old = null;
   } else {        // ... has been changed
      old = fact.set(value); fact.setFrom(location);
   }

   // Amend the primary and peers
   if (key.equals(primaryKey)) primary = fact.get();
   amendAll(fact);
   return old;
}

// Set, en masse, no frills. Do executables & special cases using regular
// set, please.
// Was synchronized, will be again one day.
public void setMany(Object keys[], Object values[]) throws IOException {
   // Kludge. FIXME later. Belongs in glyph.
   for (int j = 0; j< values.length; j++)
      if (values[j] == null) values[j] = new double[0];
   // end kludge

   ArrayEnumeration those = new ArrayEnumeration(keys, values);
   Fact big = new Fact(keys[0], those, values[0], location);
   for (int i = 0; i < big.extent(); i++) {
      Fact f = big.getPayload(i); properties.put(f,f);
   }
   amendAll(big);
}

// Get an item of this facet's data.
// Returns the value, and notifies peers if this is for a callback.
// Was synchronized, will be again one day.
public Object get(Object key, boolean forCallback) throws IOException {
   // Reality checks
   if (key == null) return (forCallback ? NO_OP : null);
   if (key.equals(primaryKey)) return primary;

   // Retrieve the fact
   Fact f = (Fact)(properties.get(key));
   if (f == null) {
      return (forCallback ? NO_OP : null);
   } else if (forCallback) {        // Execute...
      Executable fx = (Executable)f;
      // log.println("Executing " + fx);
      if (fx.isGlobal()) {         // ...everywhere if global.
         evaluateAll(key, EXECUTE);  return f.get();
      } else if (fx.isFrom(location)) {        // ...here if local.
            return f.get();
      } else {
         Name n = (Name)(peers.get(new Integer(f.from())));
         if (n == null) {               // ...nowhere?
```

```
            // log.println("Collaborative callback " + key +
                        // " for nonexistent facet " + fx.from());
         } else {        // ...at origin of callback.
            Peer p = (Peer)n.get();
            p.evaluate(key, EXECUTE);
         }
         return NO_OP;
      }
   } else {
      // log.println("Evaluating " + f);
      return f.get();
   }
}

// ***** Implementations of facet behaviour ***** //

// Register a new peer, from the Peer interface.
// Three phase algorithm. On first contact, a candidate names itself as
// referee. The member contacted broadcasts to others with itself as
// referee, and the others contact the new member with no referee.
// Dead facets do not take part.
// Was synchronized, lead to threading problems. FIXME.
public Fact add(Name candidate, Name referee)
   throws IOException {
   if (state == DEAD) return null;

   Integer there = new Integer(candidate.from());
   Name myself = toName();

   // If this candidate got a location from this node, remove the marker
   Name n = (Name)(peers.get(there));
   if (n != null && !n.isComplete()) peers.remove(there);

   // log.println(myself + " embracing " + candidate + " via " + referee);
   if (referee == null) {    // Normal update.
      peers.put(new Integer(candidate.from()),candidate);
      return null;
   } else if (referee == candidate) {     // New member, first contact.
      Fact p = precis();
      addBroadcast(peers.elements(), candidate, myself);
      peers.put(new Integer(candidate.from()),candidate);
      ((Peer)(candidate.get())).add(myself,null);
      return p;
   } else {  // New member referred by existing member.
      ((Peer)(candidate.get())).add(myself,null);
      peers.put(new Integer(candidate.from()),candidate);
      return null;
   }
}

// Call add on each peer on the given list.
protected void addBroadcast(Enumeration e, Name candidate, Name referee)
   throws IOException {
   while(e.hasMoreElements()) {
```

```
      Peer p = (Peer)(((Name)(e.nextElement())).get());
      p.add(candidate, referee);
   }
}

// Unregister a peer, from the Peer interface.
// Returns true iff the named candidate was a peer.
public synchronized boolean remove(Name candidate) throws IOException {
   Integer there = new Integer(candidate.from());
   return (peers.remove(there) != null);
}

// Attempt to adopt the given parent, ancestor, and tree, and acquire a
// backing instance using the parameters given. From the Child interface.
// Called recursively on any children, with possibly different parameters.
// Returns true if the adoption is successful.
public boolean adopt(Parent p, Ancestor a, int location, Object backing[]) {
   this.location = location;
   Name me = toName();
   if (state == ACTIVE) {        // Move existing subtree
      try {
         if ((p != null) && (p != parent)) {
            if (parent != null) parent.disown(this);
            parent = p; parent.adopt(this);
         }
         if ((a != null) && (a != ancestor)) {
            if (ancestor != null) ancestor.loose(me);
            ancestor = a; ancestor.bind(me);
         }
         mapFromTree(p, backing);
         return adoptAll(p, a, location, backing);
      } catch (Exception ex) {
         log.println(identity + " could not be moved.");
         ex.printStackTrace(log); return false;
      }
   } else if (state == INACTIVE) {        // Graft new subtree
      try {
         Name comrade = a.complete(new Name(identity));
         if (comrade.isFrom(location)) {                // Already exists
            Child twin = (Child)(comrade.get());
            this.dispose();
            return twin.adopt(p,a,location, backing);
         } else if (comrade.isComplete()) {               // Mirror already exists
            Peer friend = (Peer)(comrade.get());
            Fact summary = friend.add(me,me);
            digest(summary);
            mapFromGraft(p, backing);
         } else {                                  // New mirror needed
            primary = null;
            mapFromBud(p, backing);
         }
         this.ancestor = a; a.bind(me);
         this.parent = p; p.adopt(this);
         this.state = ACTIVE;
```

```
         return adoptAll(p, a, location, backing);
      } catch (Exception ex) {
         log.println(identity + " could not be adopted.");
         ex.printStackTrace(log); return false;
      }
   } else {        // State is dead or out-of-band. Panic.
      log.println("Attempted adoption on dead node."); return false;
   }
}

// Notify all children of the adoption of their parent
protected boolean adoptAll(Parent p, Ancestor a, int l, Object b[]) {
   boolean result = true;
   for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
      Child c = (Child)(e.nextElement());
      result &= c.adopt(this,ancestor,location, b);
   }
   return result;
}

// Remove a child from the list of children, from the Parent interface.
public synchronized boolean disown(Child c) {
   return children.removeElement(c);
}

// Add a child to the list of children, from the Parent interface.
// Returns true if the child has not previously been adopted here.
public synchronized boolean adopt(Child c) {
   if (children.indexOf(c) > -1) {
      return false;
   } else {
      children.addElement(c); return true;
   }
}

// Duplicate a graft performed on another node, from the Peer interface.
// Requires more thought to supply a meaningful details argument.
public synchronized boolean graft(MarshalledRune child) throws IOException {
   try {
      Rune x = child.unmarshal();
      return x.adopt(this, ancestor, location, null);
   } catch (Exception ex) {
      log.println("Could not duplicate graft on " + identity);
      ex.printStackTrace(log);
      return false;
   }
}

// Add a child of a given name to the tree, from the Parent interface.
// Returns the child grafted, or null on failure.
public synchronized Child graft(String mirror, Object details[]) {
   if (state == INACTIVE) {
      log.println("Attempt to graft something to an inactive parent.");
      return null;
```

```
   } else if (state == ACTIVE) {
      // Avoid loops. How do we avoid more complex loops?
      if (mirror.equals(identity)) return this;

      try {
         Name x = seek(new Name(mirror));
         if (x.isFrom(location)) {                // Already here
            Child c = (Child)(x.get());
            c.adopt(this, ancestor, location, details);
            return c;
         } else if (x.isComplete()) {        // Exists elsewhere
            Peer other = (Peer)(x.get());
            Child c = (Child)(other.toCutting().unmarshal());
            c.adopt(this, ancestor, location, details);
            return c;
         } else {                                 // Create new mirror
            Child c = newBud(mirror, details);
            c.adopt(this, ancestor, location, details);
            return c;
         }
      } catch (Exception ex) {
         log.println("Graft of " + mirror + " went awry.");
         ex.printStackTrace(log); return null;
      }
   } else {        // State is DEAD or out of band
      log.println("Attempt to graft something to a dead parent.");
      return null;
   }
}

// Return the state of this object, as defined in the Child interface.
public synchronized int status() {
   return state;
}

// Read a new Fact and react accordingly, from the Peer interface.
// Return true iff the fact had some effect.
// Needs synchronization.
public boolean amend(Fact f) throws IOException {
   // Reality checks - existence, completeness, relevance, causality
   if (f == null) return false;
   if (!f.isComplete()) return false;
   if (state == DEAD) return false;
   if (!causal(f)) { queue.addElement(f); return false; }

   // Run it past the backing instance and into the properties table.
   return (reflected(f) & added(f));
}

// Ask each peer to accept a new fact. Returns true iff all the requests
// had some effect.
// Needs synchronization.
protected boolean amendAll(Fact fact) throws IOException {
   boolean result = true;
```

```
   for (Enumeration e = peers.elements(); e.hasMoreElements(); ) {
      Peer p = (Peer)( ((Name)(e.nextElement())).get());
      result &= p.amend(fact);
   }
   return result;
}


// Ingest all causally-ready facts in queue.
public synchronized void forward() {
   int n = queue.size(), i = 0;
   while ((n > 0) && (i != n)) {
      Fact f = (Fact)(queue.elementAt(i));
      if (causal(f)) {
         reflected(f); added(f);
         queue.removeElementAt(i); n--;
      } else {
         i++;
      }
   }
}

// Impress this fact upon the properties table. Return true if the attempt
// was valid and succeeded,
protected boolean added(Fact f) {
   if (f.isMultiple()) {
      for (int i = 0; i < f.extent(); i++) {
         Fact g = f.getPayload(i); properties.put(g,g);
         // log.println("Adding payload " + g);
      }
   } else {
      properties.put(f,f);
   }
   forward();        // FIXME: possible bug site.
   return true;
}


// Evaluate some of your data, from the Peer interface.
// If it turns out to be executable data, execute it and report the
// results, otherwise just evaluate it and return that.
// Was synchronized. Will be again some day.
public Object evaluate(Object key, int m) throws IOException {
   if (key == null) {
      return null;               // We ignore bogus requests.
   } else if (key.equals(primaryKey)) {
      return primary;         // We never execute primary values.
   } else {
      Fact f = (Fact)(properties.get(key));
      if (f == null) {        // We happily run no-ops.
         return null;
      } else if (f instanceof Executable) {        // We run it here if we can.
         return (m == EXECUTE) ? executed(f.get()) : f.get();
      } else {        // If not, we just hand the original value over.
```

```
         return f.get();
      }
   }
}

// Ask the entire mirror to evaluate something. Primary use is to signal
// callback events. What shall we do with the returned results?
protected void evaluateAll(Object key, int mode) throws IOException {
   for (Enumeration e = peers.elements(); e.hasMoreElements(); ) {
      Peer p = (Peer)( ((Name)(e.nextElement())).get());
      p.evaluate(key, mode);
   }
}

// ***** Utility methods ***** //

public synchronized Name toName() {
   return new Name(identity, this, location);
}

// Create a cutting of this subtree for grafting, from the Peer interface.
public synchronized MarshalledRune toCutting() throws IOException {
    return new MarshalledRune(this);
}

public String toString() {
   return "[" + identity + "/" + location + "]";
}

// Withdraw from all mirrors and trees, and prepare for finalization.
public synchronized void dispose() {
   if (state == DEAD) {
      return;
   } else try {
      Name myself = toName();
      for (Enumeration e = peers.elements(); e.hasMoreElements(); ) {
         Peer p = (Peer)(((Name)(e.nextElement())).get());
         p.remove(myself);
      }
      for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
         Child c = (Child)(e.nextElement());
         c.dispose();
      }
      if (parent != null) parent.disown(this);
      if (ancestor != null) ancestor.loose(myself);
   } catch (Exception ex) {
      log.println("Disposal failed for " + toString());
      ex.printStackTrace(log);
   }
}

public void finalize() {
   dispose();
}
```

```
// ***** Diagnostic aids ***** //
static PrintStream log;

// Print children to log stream.
public void printChildren() { printChildren(""); }
public void printChildren(String indent) {
   if (state == DEAD) {
      log.println(indent + "The cutting " + toString()); return;
   } else if (state == INACTIVE) {
      log.println(indent + "The cutting " + toString());
   } else if (state == ACTIVE) {
      log.println(indent + "The " + toString());
   } else {
      log.println(indent + "The anomaly " + toString()); return;
   }
   if (children.size() > 0) {
      String indent1 = " " + indent;
      for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
         Rune c = (Rune)(e.nextElement());
         c.printChildren(indent1);
      }
   }
}

public void printPeers() {
   log.print(identity + " has peers: " );
   for (Enumeration e = peers.keys(); e.hasMoreElements(); ) {
      log.print((Integer)(e.nextElement()) + " ");
   }
   log.println();
}

public void printProperties() {
   log.print(identity + " has value " + primary);
   if (properties == null || properties.size() == 0) {
      log.println(" and no metadata.");
   } else {
      log.println(" and metadata:");
      Enumeration e = properties.elements();
      while (e.hasMoreElements()) {
         log.println(" " + e.nextElement());
      }
   }
}


static {
   log = System.out;        // Change to /dev/null for silence
   executor = new Executor();
   executor.setDaemon(true); executor.start();
}

// ***** Hooks - override these in any subclass ***** //
```

```
// Absorb a precis from another facet.
protected void digest(Fact f) {
   primary = f.get();
   if (reflected(f)) added(f);
}

// Provide a precis to another facet.
protected Fact precis() {
   return new Fact(identity, properties.elements(), primary, location);
}

// Impress this fact upon the backing instance. Return true if the attempt
// was valid and succeeded, false otherwise.
protected boolean reflected(Fact f) {
   return true;
}

// Execute the specified action on the backing application.
// Return the result, or null if it didn't work.
protected Object executed(Object action) {
   return "";
}

// True iff the fact is causally-ready for update.
protected boolean causal(Fact f) {
   return true;        // FIXME - compare state vectors at least!
}

// Seek to complete a name. Should be overridden by those subclasses
// capable of completing names. By default it delegates to the root.
protected Name seek(Name n) throws IOException {
   return ancestor.complete(n);
}

// Adopt a backing instance, when this facet has just been budded.
protected void mapFromBud(Parent p, Object backing[]) { }

// Adopt a backing instance, when this facet has just been grafted.
protected void mapFromGraft(Parent p, Object backing[]) { }

// Adopt a backing instance, when this facet already has one.
protected void mapFromTree(Parent p, Object backing[]) { }

// True iff the key designates the primary value
protected boolean isValueKey(Object x) {
   return x.toString().equals("Value");
}

// True iff the key designates executable metadata (callbacks)
protected boolean isExecutableKey(Object x) {
   return false;
}
```

```
   // Create a new bud.
   protected Child newBud(String name, Object details[]) throws IOException {
      return new Rune(name);
   }

   // ***** Utility classes ***** //

   class ArrayEnumeration implements Enumeration {
      Object keys[], values[]; int current;
      public ArrayEnumeration(Object keys[], Object values[]) {
         this.keys = keys; this.values = values; this.current = 0;
      }
      public boolean hasMoreElements() {
         return (current < keys.length);
      }
      public Object nextElement() {
         if (!hasMoreElements()) return null;
         Fact temp = new Fact(keys[current], values[current], location);
         current++; return temp;
      }
   }

   static class Executor extends Thread {
      Vector requests;
      boolean alive;

      public Executor() {
         super("Executor"); alive = true; requests = new Vector();
      }

      public synchronized void execute(Runnable r) {
         requests.addElement(r); if (requests.size() == 1) notify();
      }

      public synchronized void dispose() { alive = false; }

      public void run() {
         while (true) synchronized (this) {
            while (requests.size() <= 0) {
               try { wait(); } catch (InterruptedException ie) {}
            }
            while (requests.size() > 0) {
               Runnable r = (Runnable)(requests.elementAt(0));
               r.run(); requests.removeElementAt(0);
            }
            if (alive == false) return;        // Die if necessary
         }
      }
   }
}
```

Class `mirror.Root`

```
package mirror;

import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

// A root is the facet at the base of a facet tree. The mirror it belongs to
// underlies the entire forest, providing a known point of communication.
// Roots do not have parents or ancestors, and do their own naming services.
public class Root extends Rune implements Ancestor {
   int nameSource;
   Hashtable names;

   // Construct a one-node facet tree. Seek the rest of the forest on the
   // local host, then another host if supplied, using the RMI registry.
   public Root(String identity, String host) throws IOException {
      super(identity);
      String listing[] = Naming.list("///");
      String aFacet = "rmi:/facet." + identity;
      Ancestor other = null;

      // Attempt to find a listing for this forest somewhere
      try {
         for (int i = 0; i<listing.length; i++) {
            if (listing[i].startsWith(aFacet)) {
               String suffix = listing[i].substring(5);
               log.println("Found " + suffix);
               other = (Ancestor)(Naming.lookup("///"+suffix));
               break;
            }
         }

         if (other == null && host != null) {
            listing = Naming.list("//" + host + "/");
            for (int i = 0; i<listing.length; i++) {
               if (listing[i].startsWith(aFacet)) {
                  String suffix = listing[i].substring(5);
                  log.println("Found " + suffix);
                  other = (Ancestor)(Naming.lookup("///"+suffix));
                  break;
               }
            }
         }
      } catch (Exception ex) {
         log.println("RMI search failed.");
         ex.printStackTrace(log);
         other = null;
      }

      if (other == null) {
         location = 0; primary = new Integer(0);
```

```
      nameSource = 0; names = new Hashtable(); state = ACTIVE;
   } else {
      primary = other.freshIdentifier(this);
      location = ((Integer)primary).intValue();
      Name me = toName(); digest(other.add(me, me));
      nameSource = 0; names = new Hashtable(); state = ACTIVE;
   }

   mapFromRoot();
   Naming.rebind("///facet." + identity + "." + location, this);
}


// ***** Implementation of the Ancestor interface ***** //

// Add a binding to the distributed registry
public boolean bind(Name binding) throws IOException {
   if (binding.isComplete()) {
      // FIXME - check for age?
      names.put(binding, binding); return true;
   } else {
      return false;
   }
}

// Remove a binding from the distributed registry
public boolean loose(Name binding) throws IOException {
   return (names.remove(binding) != null);
}

public Name complete(Name binding) throws IOException {
   return (binding.complete(peers, names));
}

public Object freshIdentifier(Object context) throws IOException {
   if (context == null) {
      return location + "." + (nameSource++);
   } else if (context instanceof Ancestor) {
      Integer result = freshLocation();
      peers.put(result, new Name(identity));
      return result;
   } else {
      return location + "." + (nameSource++);
   }
}

// Fetch a fresh location number in the range 0..maxnodes, by treating
// the name space as a branching-ary tree. This location may allocate any
// vacant children it has, and if all children are instantiated, it
// delegates the request to one of them. On overflow, start at node 0.
// If no node 0...(-:
static final int branching = 3;
static final int maxnodes = 27;
protected Integer freshLocation() throws IOException {
```

```
   synchronized (peers) {
      int base = location * branching + 1;
      if (base >= maxnodes) {                // Wrap around if too big
         Object n = peers.get(new Integer(0));
         if (n == null) return (new Integer(0));
         return (Integer)(((Ancestor)n).freshIdentifier(this));
      }
      // Seek local answer to query
      for (int offset = 0; offset < branching; offset++) {
         Integer x = new Integer(base + offset);
         if (peers.get(x) == null) return x;
      }
      // Delegate to a descendant
      for (int offset = 0; offset < branching; offset++) {
         Object n = peers.get(new Integer(base + offset));
         Object x = ((Ancestor)n).freshIdentifier(this);
         if (x != null) return (Integer)x;
      }
      return null;
   }
}


// ***** Utility methods ***** //

public synchronized Name toName() {
   return new Name(identity, this, location);
}

public String toString() {
   return "root of " + identity + " at " + location;
}

// Withdraw from all mirrors and trees, and prepare for finalization.
public synchronized void dispose() {
   super.dispose();
   try {
      Naming.unbind("///facet." + identity + "." + location);
   } catch (Exception ex) {
      log.println("Cannot unbind myself.");
      ex.printStackTrace(log);
   }
}

// ***** Hooks - override these in any subclass ***** //

// Absorb a precis from another facet.
protected void digest(Fact f) {
   primary = new Integer(location); if (reflected(f)) added(f);
}

// Provide a precis to another facet.
protected Fact precis() {
   return new Fact(identity, properties.elements(), primary, NO_LOCATION);
```

```
   }

   // Seek to complete a name.
   protected Name seek(Name n) throws IOException { return complete(n); }

   // Attach this root to the backing application in the first instance.
   protected boolean mapFromRoot() {
      return true;
   }
}
```

B.2 Fact hierarchy

Class `mirror.Fact`

```
package mirror;

import java.util.*;
import java.io.*;

// A Fact is a binding between a key and a value, with an optional payload
// and origin. The payload is a vector of Facts, the origin is an integer
// location. Fact values are mutable; payloads, keys, and locations are not.

public class Fact implements Serializable, Cloneable {
   Object key;
   Object value;
   Vector payload;
   int location;

   // Create a singular fact.
   public Fact(Object key, Object value, int location) {
      this.key = key; this.value = value;
      this.payload = null; this.location = location;
   }

   // Create a multiple fact.
   public Fact(Object key, Enumeration stuff, Object value, int location) {
      this.key = key; this.value = value; this.location = location;
      this.payload = new Vector();        // Capacity increment 1?
      while (stuff.hasMoreElements())
         this.payload.addElement(stuff.nextElement());
   }

   public int from() { return location; }
   public boolean isComplete() { return (value != null); }
   public boolean isMultiple() { return (payload != null); }
   public boolean isFrom(int here) { return here == location; }

   public Object getKey() { return key; }
   public Object get() { return value; }
```

```
public int extent() { return isMultiple() ? payload.size() : 0; }
public Fact getPayload(int ix) {
   return isMultiple() ? (Fact)(payload.elementAt(ix)) : null;
}

public Object set(Object value) {
   Object old = this.value; this.value = value; return old;
}
public int setFrom(int l) {
   int old = location; location = l; return l;
}

// Multiple facts respect pointer equality.
// Singular facts respect key equality.
// Everything else compares with the key.
public boolean equals(Object x) {
   if (x instanceof Fact) {
      Fact f = (Fact)x;
      return f.isMultiple() ? (f == this) : f.getKey().equals(key);
   } else {
      return x.equals(key);
   }
}

// hashCode() is consistent with equals.
public int hashCode() {
   return isMultiple() ? super.hashCode() : key.hashCode();
}

// Multiple facts are listed on cardinality.
// Singular facts are listed as a tuple.
public String toString() {
   if (isMultiple()) return "an "+extent()+"-ary fact";
   return "(" + key + "<=>" + value + "[" + location + "])";
}

/*
// Medium deep copy. Not unless everything is Cloneable.
public Object clone() {
   Object newvalue = (value == null) ? null : value.clone();
   if (payload == null) {
      return new Fact(key.clone(), newvalue, location);
   } else {
      return new Fact(key.clone(), payload.elements(), newvalue, location);
   }
}
*/

// Shallow copy. DANGEROUS.
public Object clone() {
   if (payload == null) {
      return new Fact(key, value, location);
   } else {
```

```
         return new Fact(key, payload.elements(), value, location);
      }
   }
}
```

Class `mirror.Name`

```
package mirror;

import java.util.*;
import java.io.*;

// A Name is a reference on steroids, binding a name to a rune. Names are
// always singular, but not always complete. Names can complete themselves.
public class Name extends Fact {
   int state;

   // Possible states
   static final int NOMINAL = 0;
   static final int SEARCHING = 1;
   static final int FOUND = 2;

   // Use the supplied information to complete yourself.
   public Name complete(Hashtable peers, Hashtable reg) throws IOException {
      if (isComplete()) {
         state = NOMINAL; // We already gots a referent, than' you.
      } else if (state == NOMINAL) {    // Checking out at home
         Name n = (Name)(reg.get(this));
         if ((n != null) && (n.isComplete())) {
            state = NOMINAL; this.value = n.get();
            this.location = n.from();
         } else {
            state = SEARCHING;
            for (Enumeration e = peers.elements(); e.hasMoreElements(); ) {
               n = ((Ancestor)((Name)(e.nextElement())).get()).complete(this);
               if (n.isComplete()) {
                  state = NOMINAL; this.value = n.get();
                  this.location = n.from(); break;
               }
            }
         }
      } else if (state == SEARCHING) {  // Checking out away from home
         Name n = (Name)(reg.get(this));
         if (n != null && n.isComplete()) {
            state = NOMINAL; this.value = n.get();
            this.location = n.from();
         }
      }
      return this;
   }

   public String toString() {
```

```
      return key + " [" + location + "]";
   }

   public Name(String which, Object what, int where) {
      super(which, what, where); this.state = NOMINAL;
   }

   public Name(String which) {
      super(which, null, Rune.NO_LOCATION);
   }
}
```

Class `mirror.Executable`

```
package mirror;

// A fact whose contents are meant to be run. Contains an alternate
// definition in case it's meant to be run in several places.
public class Executable extends Fact {
   Object alternate;
   boolean global;

   public Executable(Object key, Object value, int location) {
      super(key, value, location);
      this.alternate = value; this.global = false;
   }

   public Executable(Object key, Object value, Object alternate, int l) {
      super(key, value, l);
      this.alternate = alternate; this.global = false;
   }

   public boolean isGlobal() { return global; }
   public Object get() { return global ? alternate : value; }
   public void setGlobal(boolean yah) { global = yah; }

   public String toString() {
      return super.toString() + "(X)";
   }
}
```

B.3 Matlab bindings

Class `mirror.Glyph`

This is a subclass of `mirror.Rune` with the appropriate hooks overridden to issue Matlab commands via and instance of `com.mathworks.jmi.Matlab`. Threading

problems described in section 3.3.3 have forced the splitting of `graft` into several methods.

```
package mirror;

import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import com.mathworks.jmi.*;

// A Glyph is a Rune backed by a Matlab Handle Graphics object.
public class Glyph extends Rune {
   transient double handle;        // Handle of the backing instance
   static Matlab matlab;        // Interface to the backing application
   String kind;                        // Constructor for the backing instance
   boolean isUIControl;                // true iff this is a user interface control

   // Wildcard or unallocated handle
   static double NO_HANDLE = -3.1415926483;

   // Construct an inactive facet of this HG object
   public Glyph(String mirror) throws IOException {
      super(mirror); this.handle = NO_HANDLE; this.kind = null;
   }

   // Deserialize an inactive facet of this HG object
   private void readObject(ObjectInputStream stream)
      throws IOException, ClassNotFoundException {
      stream.defaultReadObject();
      this.handle = NO_HANDLE;
   }

   // ***** Layered reimplementation of grafting and adopting ***** //

   // This is the root of an inactive subtree. Passes back the information
   // for the Matlab bridging code to install callbacks and metadata. First
   // entry is tag, second entry is kind, third is keys, fourth is values,
   // fifth is this glyph, sixth through whatever are similar entries
   // for children.
   Object[] subtreeDetails() {
      Object[] result = new Object[children.size() + 5];
      result[0] = identity; result[1] = kind;
      Object keys[] = new Object[properties.size()];
      Object values[] = new Object[properties.size()];
      Enumeration e = properties.elements();
      for (int i = 0; e.hasMoreElements(); i++) {
           Fact f = (Fact)(e.nextElement());
           keys[i] = f.getKey(); values[i] = f.get();
      }
      result[2] = keys; result[3] = values;
      result[4] = this;
```

```
   e = children.elements();
   for (int i = 0; e.hasMoreElements(); i++) {
        Glyph g = (Glyph)(e.nextElement());
        result[i+5] = g.subtreeDetails();
   }
   return result;
}

// The Name refers to a facet of a mirror elsewhere. Obtain the cutting,
// extract details for Matlab to create things on. Adoption happens in the
// other stage, after the backing instances exist.
public Object[] graftStart(Name that)
   throws ClassNotFoundException, IOException {
   Peer p = (Peer)(that.get());
   Glyph g = (Glyph)(p.toCutting().unmarshal());
   return g.subtreeDetails();
}

public synchronized Child graft(String mirror, Object details[]) {
   log.println("Conventional grafting does not work with these.");
   return null;
}

// The Glyph is the root of an inactive subtree, the backing instances are
// in position and prepped. Take the handles, do the adoption. First entry
// is a double[1], this handle, second through whatever are similar
// entries for each child.
public void graftFinish(Glyph x, Object handles[]) throws IOException {
   x.graftAdopt(this, ancestor, location, handles);
}

public void graftAdopt(Parent p, Ancestor a, int l, Object handles[])
   throws IOException {
   // Generic paperwork - ERROR CHECKING IS SOMEONE ELSE'S PROBLEM
   this.location = l; this.parent = p; p.adopt(this);
   Name n = a.complete(new Name(identity));
   Name me = toName();
   this.ancestor = a; a.bind(me);
   Peer other = (Peer)(n.get());
   other.add(me, me);

   // Bind backing instance to first handle
   Double myH = (Double)(handles[0]);
   this.handle = myH.doubleValue();

   // Adopt each child recursively
   // This hack depends critically on successive Enumerations yielding the
   // same traversal order. If not, whack tags in as well and do a search.
   Enumeration e = children.elements();
   for (int i = 1; e.hasMoreElements(); i++) {
      Object theirH[] = (Object [])(handles[i]);
      Glyph g = (Glyph)(e.nextElement());
      g.graftAdopt(this, a, l, theirH);
   }
```

```
   state = ACTIVE;
}

// Create a new bud, point it at an existing HG object, get it adopted.
// Note: EXISTING MIRRORS ARE SOMEONE ELSE'S PROBLEM.
public Child budMatlab(String that, double handle, String kind,
   Object keys[], Object values[]) throws IOException {
   Object details[] = {kind, keys, values, new Double(handle)};
   Glyph g = new Glyph(that);
   g.adopt(this, ancestor, location, details);
   return g;
}

// Seek out a Name referring to a mirror Matlab wants a facet of
public Name seek(String identifier) throws IOException {
   return seek(new Name(identifier));
}

protected boolean isUIComponent(String k) {
   if (kind.equalsIgnoreCase("Uicontrol")) return true;
   if (kind.equalsIgnoreCase("Uimenu")) return true;
   if (kind.equalsIgnoreCase("Uicontextmenu")) return true;
   return false;
}


// ***** Utility methods ***** //

public String toString() {
   return "facet of " + kind + " " + handle + " at " + location;
}

// ***** Hooks - override these in any subclass ***** //

// Provide a precis to another facet.
protected Fact precis() {
   return new Fact(identity, properties.elements(), primary, location);
}

// Impress this fact upon the backing instance. Return true if the attempt
// was valid and succeeded, false otherwise.
protected boolean reflected(Fact f) {
   // Executable metadata is held exclusively within the rune
   if (f instanceof Executable) return true;

   // Dump multiple values straight in
   if (f.isMultiple()) try {
      Object keys[] = new Object[f.extent()];
      Object values[] = new Object[f.extent()];
      for (int i = 0; i < f.extent(); i++) {
         Fact g = (Fact)(f.getPayload(i));
         // log.println("Reflecting payload " + g);
         // FIXME: don't when isExecutableKey(keys[i])!
```

```
            keys[i] = g.getKey(); values[i] = g.get();
        }
        Object args[] = {new Double(handle), keys, values};
        // FIXME - change to asynchronous execution?
        // double[] d = (double[])(matlab.feval("stdset",args));
        // return (d[0] == 1);
        matlab.feval("stdset",args,null); return true;
    } catch (Exception ex) {
        log.println("Reflection failed on " + f);
        ex.printStackTrace(log); return false;
    }

    // Change the key if it is a primary value
    if (f.getKey().equals(primaryKey)) {
        if (isUIControl) {     // UIControl has a primary value
            f = new Fact(primaryKey,f.get(), NO_LOCATION);
        } else {
            return true;        // Other HG objects don't have one?
        }
    }

    // Must be a single key/value pair with correct key - punch it in
    try {
        Object args[] = {new Double(handle), f.getKey(), f.get()};
        // FIXME: change to asynchronous execution?
        double[] d = (double[])(matlab.feval("stdset",args));
        return (d[0] == 1);
    } catch (Exception ex) {
        log.println("Reflection failed on " + f);
        ex.printStackTrace(log); return false;
    }
}

// Execute the specified action on the backing application.
// Return the result, or null if it didn't work.
// Should we go asynchronous?
protected Object executed(Object action) {
    try {
        // log.println(location + ": executing " + action);
        Object x = matlab.eval(action.toString());
        return x;
    } catch (Exception ex) {
        log.println("Execution failed on " + action);
        ex.printStackTrace(log); return null;
    }
}

// True iff the fact is causally-ready for update.
protected boolean causal(Fact f) {
    return true;        // FIXME - compare state vectors at least!
}

// Adopt a backing instance, when this facet has just been budded.
protected void mapFromBud(Parent p, Object backing[]) {
```

```
      // See budMatlab for details.
      this.kind = (String)backing[0];
      this.isUIControl = isUIComponent(this.kind);
      this.handle = ((Double)(backing[3])).doubleValue();
      Object keys[] = (Object[])(backing[1]);
      Object values[] = (Object[])(backing[2]);
      for (int i = 0; i<keys.length; i++) {
         if (values[i] == null) values[i] = new double[0];
         Fact f = isExecutableKey(keys[i])
                ? new Executable(keys[i], values[i], location)
                : new Fact(keys[i], values[i], location);
         properties.put(f,f);
      }
   }

   // Adopt a backing instance, when this facet has just been grafted.
   protected void mapFromGraft(Parent p, Object backing[]) {
      // Deprecated - see graftStart for details
   }

   // Adopt a backing instance, when this facet already has one.
   protected void mapFromTree(Parent p, Object backing[]) {
      // Deprecated - see graftStart for details
   }

   // True iff the key designates the primary value
   protected boolean isValueKey(Object x) {
      String key = x.toString();
      return key.equals("Value");        // FIXME - needs to be different
   }

   // True iff the key designates executable metadata (callbacks)
   protected boolean isExecutableKey(Object x) {
      String s = x.toString();
      if (s.equalsIgnoreCase("deletefcn")) return true;
      if (s.equalsIgnoreCase("createfcn")) return true;
      if (s.equalsIgnoreCase("buttondownfcn")) return true;
      if (s.equalsIgnoreCase("callback")) return true;
      if (s.equalsIgnoreCase("windowbuttondownfcn")) return true;
      if (s.equalsIgnoreCase("windowbuttonupfcn")) return true;
      if (s.equalsIgnoreCase("windowbuttonmotionfcn")) return true;
      if (s.equalsIgnoreCase("resizefcn")) return true;
      if (s.equalsIgnoreCase("closerequestfcn")) return true;
      if (s.equalsIgnoreCase("keypressfcn")) return true;
      if (s.equalsIgnoreCase("resizefcn")) return true;
      return false;
   }

   static {
      matlab = new Matlab();
   }
}
```

Class `mirror.RootWindow`

This is a subclass of `mirror.Root` which uses as backing instance the Matlab root window. The same issues as for `mirror.Glyph` have been addressed, mostly identically.

```
package mirror;

import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import com.mathworks.jmi.*;

// A Root facet backed by a Matlab session. Meant to be started from within
// the Matlab session using their Java API.
public class RootWindow extends Root {
   protected static Matlab matlab;      // The access point to the Matlab VM
   protected static Executor executor;  // An objective point of view

   // Attach this object to the Java root window, putting the keys and values
   // in metadata.
   public RootWindow(String identity, String host,
      Object keys[], Object values[]) throws IOException {
      super(identity, host);
      // if (host != "ack") return;

      if (keys != null) {
         for (int i = 0; i < keys.length; i++) {
            if (values[i] == null) values[i] = new double[0];
            Fact f = new Fact(keys[i],values[i],location);
            properties.put(f,f);
         }
      }
   }

   // ***** Layered reimplementation of grafting and adopting ***** //

   // The Name refers to a facet of a mirror elsewhere. Obtain the cutting,
   // extract details for Matlab to create things on. Adoption happens in the
   // other stage, after the backing instances exist.
   public Object[] graftStart(Name that)
      throws ClassNotFoundException, IOException {
      Peer p = (Peer)(that.get());
      Glyph g = (Glyph)(p.toCutting().unmarshal());
      return g.subtreeDetails();
   }

   public void graftFinish(Glyph x, Object handles[]) throws IOException {
      x.graftAdopt(this, this, location, handles);
   }
```

```
   public synchronized Child graft(String mirror, Object details[]) {
      log.println("Conventional grafting does not work with these.");
      return null;
   }

   // Create a new bud, point it at an existing HG object, get it adopted.
   // Note: EXISTING MIRRORS ARE SOMEONE ELSE'S PROBLEM.
   public Child budMatlab(String that, double handle, String kind,
      Object keys[], Object values[]) throws IOException {
      Object details[] = {kind, keys, values, new Double(handle)};
      Glyph g = new Glyph(that);
      g.adopt(this, this, location, details);
      return g;
   }

   // Seek out a Name referring to a mirror Matlab wants a facet of
   public Name seek(String identifier) throws IOException {
      return seek(new Name(identifier));
   }

// STOP HERE

   // ***** Utility methods ***** //

   public String toString() {
      return "root window of session " + identity + " [" + location + "]";
   }

   // ***** Hooks - override these in any subclass ***** //

   // Impress this fact upon the backing instance. Return true if the attempt
   // was valid and succeeded, false otherwise.
   protected boolean reflected(Fact f) {
      // Reject multiple value straight off
      if (f == null) {
         log.println("Reflecting a null fact."); return false;
      }

      // Dump multiple values straight in
      if (f.isMultiple()) try {
         Object keys[] = new Object[f.extent()];
         Object values[] = new Object[f.extent()];
         for (int i = 0; i < f.extent(); i++) {
            Fact g = (Fact)(f.getPayload(i));
            keys[i] = g.getKey(); values[i] = g.get();
         }
         Object args[] = {new Double(0), keys, values};
         // FIXME - change to asynchronous execution?
         double[] d = (double[])(matlab.feval("stdset",args));
         return (d[0] == 1);
      } catch (Exception ex) {
         log.println("Reflection failed on " + f);
         ex.printStackTrace(log); return false;
```

```
   }

   // Change the key if it is a primary value
   if (f.getKey().equals(primaryKey)) {
      return true;        // We do not amend the value of a root
   }

   // Must be a single key/value pair with correct key - punch it in
   try {
      log.println("Reflecting " + f.getKey());
      Object args[] = {new Double(0), f.getKey(), f.get()};
      // FIXME: change to asynchronous execution?
      double[] d = (double[])(matlab.feval("stdset",args));
      return (d[0] == 1);
   } catch (Exception ex) {
      log.println("Reflection failed on " + f);
      ex.printStackTrace(log); return false;
   }
}

// Execute the specified action on the backing application.
// Return the result if it worked, null if it didn't.
protected Object executed(Object action) {
   try {
      return matlab.eval(action.toString());
   } catch (Exception ex) {
      log.println("Execution failed on " + action);
      ex.printStackTrace(log); return null;
   }
}

// Digest a precis of the mirror
protected void digest(Fact f) {
   primary = new Integer(location);
   Object keys[] = new Object[f.extent()];
   Object values[] = new Object[f.extent()];
   for (int i = 0; i<f.extent(); i++) {
      Fact g = f.getPayload(i);
      keys[i] = g.getKey(); values[i] = g.get();
      properties.put(g,g);
   }
   Object args[] = {new Double(0), keys, values};
   matlab.feval("stdset",args,null);
}

// True iff the key designates the primary value
protected boolean isValueKey(Object x) {
   return false;
   // x.toString().equalsIgnoreCase(""); // FIXME - eh?
}

// True iff the key designates executable metadata (callbacks)
protected boolean isExecutableKey(Object x) {
   return false; // Nothing a Matlab root window can do is executable
```

```
   }

   // Create a new bud for this tree
   protected Child newBud(String mirror) throws IOException {
      return new Glyph(mirror);
   }

   // Attach this root to the backing application in the first instance.
   protected boolean mapFromRoot() {
      // What goes here?
      return true;
   }

   static {
      matlab = new Matlab();
   }
}
```

Matlab definitions for class `double`

These are the Matlab functions which replace the basic Handle Graphics commands in Concurrent Matlab over RMI. They override these by virtue of being placed in a directory designated to hold definitions for class `double`, ordinary numbers. Some few of these commands also have forms where none of the arguments are a number. These are duplicated in the directory for class `char`, ordinary strings. The files each start with a comment of the form `%CLASS/METHOD This method does....`

```
%DOUBLE/SET        Set object properties.
%   This version knows about glyphs.


function x = set(hndl,varargin)

if (nargout > 0),
   x = builtin('set',hndl,varargin{:});
else,
   builtin('set',hndl,varargin{:});
end;

[k,v,n] = setargs(varargin{:}); if (n == 0), return; end;

% Check each handle for glyphs; notify them if found
for (h = hndl(:)'),
   r = getappdata(h,'Rune');
```

```
   if (~isempty(r)),
      nlumped = 0;
      for (i = 1:n),
         k{i} = lower(k{i}); followthrough = 1;
         %disp(['Setting ' k{i}]);
         switch (k{i}),
            case {'xdata','ydata','zdata','cdata',...
                  'faces','vertices','vertexnormals','facevertexcdata'},
               nlumped = nlumped + 1;
               lKeys{nlumped} = k{i}; lValues{nlumped} = v{i};
               followthrough = 0;
            case 'parent',        % FIXME HARD
               parent = getappdata(v{i}, 'Rune');
               location = getappdata(0, 'RuneLocation');
               ancestor = getappdata(0, 'Rune');
               r.adopt(parent, ancestor, location);
               followthrough = 0;
            case 'tag',
               % Changing the mirror identifier? No.
               error('Don''t do that.');
            case {'currentfigure','showhiddenhandles'},
               followthrough = 0;        % Bug fix
            case 'deletefcn',
               stdset(h,'DeleteFcn','RuneDelete');
            case 'buttondownfcn',
               stdset(h,'ButtonDownFcn','RuneButtonDown');
            case 'callback',
               stdset(h,'Callback','RuneCallback');
            case 'resizefcn',
               stdset(h,'ResizeFcn','RuneResize');
            case 'keypressfcn',
               stdset(h,'KeyPressFcn','RunePress');
            case 'closerequestfcn',
               stdset(h,'CloseRequestFcn','RuneCloseReq');
            case 'windowbuttondownfcn',
               stdset(h,'WindowButtonDownFcn','RuneWindowDown');
            case 'windowbuttonupfcn',
               stdset(h,'WindowButtonUpFcn','RuneWindowUp');
            case 'windowbuttonmotionfcn',
               stdset(h,'WindowButtonMotionFcn','RuneWindowMotion');
         end;
         if (followthrough), r.set(k{i}, v{i}); end;
      end;
      if (nlumped > 0),
         r.setMany(lKeys, unempty(lValues));
      end;
   end;
end;



%DOUBLE/GET        Get handle graphics properties.
```

```
function out = get(hndls, varargin);
out = builtin('get',hndls, varargin{:});



%DOUBLE/DELETE   Delete shared graphics object.
%   If the argument is a handle, and the object is shared, it will be
%   unshared before being deleted. Otherwise, as DELETE.
function delete(h)
builtin('delete',h);



%DOUBLE/AXES   Create axes in arbitrary positions.
%   If the parent figure is shared, and the axes are tagged, the axes will
%   also be shared. Otherwise, as AXES.

function h = axes(varargin)

% Determine whether the axes already exist
if (nargin == 0),
   isnew = 1;
elseif (~ishandle(varargin{1}))
   isnew = 1;
else,
   isnew = 0;
end;


% Run the original command
if (nargin == 1),
   builtin('axes',varargin{:});                % Shift focus
   hndl = varargin{1};
else
   hndl = builtin('axes',varargin{:});        % Create/amend axes
   h = hndl;
end;

phndl = builtin('get',hndl,'Parent');

% If the parent is a glyph, and the new child is named, make it a glyph
if (isnew),
   if (isrune(phndl)),
      tag = get(hndl,'Tag');
      if (~isempty(tag)),
         parent = getappdata(phndl,'Rune');
         name = parent.seek(tag);
         location = getappdata(0,'RuneLocation');
         if (~name.isComplete),                        % Totally new axes
            keys = lower(fieldnames(builtin('set',hndl)));
            values = unempty(builtin('get',hndl,keys)');
            glyph = parent.budMatlab(tag, hndl, 'axes', keys, values);
```

```
            setappdata(hndl,'Rune',glyph); runify(hndl);
         elseif (name.isFrom(location)),        % Existing axes
            builtin('delete',hndl);
            hndl = findobj(0,'tag',tag,'type','axes');
            return;
         else,                                          % Grafted axes
            details = parent.graftStart(name);
            details = unmarshal(details);
            handles = runify(hndl, 0, details);
            glyph = details{5};
            parent.graftFinish(glyph, handles);
         end;
      end;
   end;
end;

if (nargout > 0), h = hndl; end;



%DOUBLE/FIGURE Create figure window.
%   If the root is shared, and the figure is tagged, it will be shared.
%   Otherwise, as FIGURE.

function h = figure(varargin)

% Determine whether the figure already exists
if (nargin == 0),
   isnew = 1;
elseif (~ishandle(varargin{1}))
   isnew = 1;
else,
   isnew = 0;
end;

% Run the original command
hndl = builtin('figure',varargin{:});

% If the parent is a glyph, and the new child is named, make it a glyph
if (isnew),
   if (isrune(0)),
      tag = get(hndl,'Tag');
      if (~isempty(tag)),
         parent = getappdata(0,'Rune');
         name = parent.seek(tag);
         location = getappdata(0,'RuneLocation');
         if (~name.isComplete),                % First facet of mirror
            keys = lower(fieldnames(builtin('set',hndl)));
            values = unempty(get(hndl,keys)');
            glyph = parent.budMatlab(tag,hndl, 'figure', keys, values);
            setappdata(hndl,'Rune',glyph);
            runify(hndl);
         elseif (name.isFrom(location)),
            builtin('delete',hndl);        % FIXME HARD
```

```
            hndl = findobj(0,'tag',tag,'type','figure');
            return;
         else,                                % Grafting from elsewhere
            details = parent.graftStart(name);
            details = unmarshal(details);
            handles = runify(hndl, 0, details);
            glyph = details{5};
            parent.graftFinish(glyph, handles);
         end;
      end;
   end;
end;

if (nargout > 0),
   h = hndl;
end;
```

```
%DOUBLE/IMAGE        Display image.
%   If the parent axes are shared, so will the image be. If the image is
%   untagged, one will be provided. Otherwise, as IMAGE.

function h = image(varargin)

% Run the original command
hndl = builtin('image',varargin{:});
phndl = builtin('get',hndl,'Parent');
root = getappdata(0,'Rune');

% If the parent axes are a glyph, make the child a glyph
if (isappdata(phndl,'Rune')),
   tag = get(hndl,'tag'); parent = getappdata(phndl,'Rune');
   parent = getappdata(phndl,'Rune');
   if (isempty(tag)),        % generate a name if not given one
      tag = root.freshIdentifier([]); tag = tag.toString;
      builtin('set',hndl,'tag',tag);
   end;
   name = parent.seek(tag);
   location = getappdata(0,'RuneLocation');
   if (~name.isComplete),        % Totally new image
      keys = lower(fieldnames(builtin('set',hndl)));
      values = unempty(builtin('get',hndl,keys)');
      glyph = parent.budMatlab(tag, hndl, 'image', keys, values);
      setappdata(hndl, 'Rune',glyph); runify(hndl);
   elseif (name.isFrom(location)),        % Existing image - FIXME
      builtin('delete',hndl);
      hndl = findobj(0,'tag',tag,'type','image');
      return;
   else,                        % Grafted image
      details = parent.graftStart(name);
      details = unmarshal(details);
```

```
      handles = runify(hndl, 0, details); glyph = details{5};
      parent.graftFinish(glyph, handles);
   end;
end;

% Return something if required
if (nargout > 0),
   h = hndl;
end;
```

```
%DOUBLE/LIGHT        Create light.
%   If the parent axes are shared, then so will this be. If a tag is not
%   supplied, one will be generated. Otherwise, as LIGHT.

function h = light(varargin)

% Run the original command
hndl = builtin('light',varargin{:});
phndl = builtin('get',hndl,'Parent');
root = getappdata(0,'Rune');

% If the parent axes are a glyph, make the child a glyph
if (isappdata(phndl,'Rune')),
   tag = get(hndl,'tag'); parent = getappdata(phndl,'Rune');
   parent = getappdata(phndl,'Rune');
   if (isempty(tag)),        % generate a name if not given one
      tag = root.freshIdentifier([]); tag = tag.toString;
      builtin('set',hndl,'tag',tag);
   end;
   name = parent.seek(tag);
   location = getappdata(0,'RuneLocation');
   if (~name.isComplete),        % Totally new light
      keys = lower(fieldnames(builtin('set',hndl)));
      values = unempty(builtin('get',hndl,keys)');
      glyph = parent.budMatlab(tag, hndl, 'light', keys, values);
      setappdata(hndl, 'Rune',glyph); runify(hndl);
   elseif (name.isFrom(location)),        % Existing light - FIXME
      builtin('delete',hndl);
      hndl = findobj(0,'tag',tag,'type','light');
      return;
   else,                         % Grafted light
      details = parent.graftStart(name);
      details = unmarshal(details);
      handles = runify(hndl, 0, details); glyph = details{5};
      parent.graftFinish(glyph, handles);
   end;
end;

% Return something if required
if (nargout > 0),
```

```
   h = hndl;
end;
```

```
%DOUBLE/LINE         Create line.
%   If the parent axes is shared, the line will also be shared. If the line
%   has no tag, one will be provided. Otherwise, as LINE.

function h = line(varargin)

% Run the original command and parse the output
hndls = builtin('line',varargin{:});
phndls = builtin('get',hndls,'Parent');
nlines = length(hndls);
if (nlines > 1), phndls = cat(1,phndls{:}); end;
root = getappdata(0,'Rune');

% For each line, if the parent is a glyph, make the child a glyph
for (i = 1:nlines),
   if (isappdata(phndls(i),'Rune')),
      tag = get(hndls(i),'Tag');
      parent = getappdata(phndls(i),'Rune');
      if (isempty(tag)),        % generate a name if not given one
         tag = root.freshIdentifier([]); tag = tag.toString;
         builtin('set',hndls(i),'tag',tag);
      end;
      name = parent.seek(tag);
      location = getappdata(0,'RuneLocation');
      if (~name.isComplete),                        % Totally new line
         keys = lower(fieldnames(builtin('set',hndls(i))));
         values = unempty(builtin('get',hndls(i),keys)');
         glyph = parent.budMatlab(tag, hndls(i), 'line', keys, values);
         setappdata(hndls(i),'Rune',glyph); runify(hndls(i));
      elseif (name.isFrom(location)),        % Existing line - FIXME
         builtin('delete',hndls(i));
         hndls(i) = findobj(0,'tag',tag,'type','line');
         return;
      else,                                          % Grafted line
         details = parent.graftStart(name);
         details = unmarshal(details);
         handles = runify(hndls(i), 0, details);
         glyph = details{5};
         parent.graftFinish(glyph, handles);
      end;
   end;
end;

% Return something if required
if (nargout > 0),
   h = hndls;
end;
```

```
%DOUBLE/PATCH        Create patch.
%   If the parent axes are shared, so will the patch be. If the patch is
%   untagged, one will be provided. Otherwise, as PATCH.

function h = patch(varargin)

% Run the original command
hndl = builtin('patch',varargin{:});
phndl = builtin('get',hndl,'Parent');
root = getappdata(0,'Rune');

% If the parent axes are a glyph, make the child a glyph
if (isappdata(phndl,'Rune')),
   tag = get(hndl,'tag'); parent = getappdata(phndl,'Rune');
   parent = getappdata(phndl,'Rune');
   if (isempty(tag)),        % generate a name if not given one
      tag = root.freshIdentifier([]); tag = tag.toString;
      builtin('set',hndl,'tag',tag);
   end;
   name = parent.seek(tag);
   location = getappdata(0,'RuneLocation');
   if (~name.isComplete),        % Totally new patch
      keys = lower(fieldnames(builtin('set',hndl)));
      values = unempty(builtin('get',hndl,keys)');
      glyph = parent.budMatlab(tag, hndl, 'patch', keys, values);
      setappdata(hndl, 'Rune',glyph); runify(hndl);
   elseif (name.isFrom(location)),        % Existing patch - FIXME
      builtin('delete',hndl);
      hndl = findobj(0,'tag',tag,'type','patch');
      return;
   else,                         % Grafted patch
      details = parent.graftStart(name);
      details = unmarshal(details);
      handles = runify(hndl, 0, details); glyph = details{5};
      parent.graftFinish(glyph, handles);
   end;
end;

% Return something if required
if (nargout > 0),
   h = hndl;
end;
```

```
%DOUBLE/RECTANGLE   Create rectangle, rounded-rectangle, or ellipse.
%   If the parent axes are shared, then the rectangle will be also. If the
%   rectangle is not tagged, one will be provided. Otherwise, as RECTANGLE.

function h = rectangle(varargin)

% Run the original command
hndl = builtin('rectangle',varargin{:});
phndl = builtin('get',hndl,'Parent');
root = getappdata(0,'Rune');

% If the parent axes are a glyph, make the child a glyph
if (isappdata(phndl,'Rune')),
   tag = get(hndl,'tag'); parent = getappdata(phndl,'Rune');
   parent = getappdata(phndl,'Rune');
   if (isempty(tag)),        % generate a name if not given one
      tag = root.freshIdentifier([]); tag = tag.toString;
      builtin('set',hndl,'tag',tag);
   end;
   name = parent.seek(tag);
   location = getappdata(0,'RuneLocation');
   if (~name.isComplete),        % Totally new rectangle
      keys = lower(fieldnames(builtin('set',hndl)));
      values = unempty(builtin('get',hndl,keys)');
      glyph = parent.budMatlab(tag, hndl, 'rectangle', keys, values);
      setappdata(hndl, 'Rune',glyph); runify(hndl);
   elseif (name.isFrom(location)),        % Existing rectangle - FIXME
      builtin('delete',hndl);
      hndl = findobj(0,'tag',tag,'type','rectangle');
      return;
   else,                         % Grafted rectangle
      details = parent.graftStart(name);
      details = unmarshal(details);
      handles = runify(hndl, 0, details); glyph = details{5};
      parent.graftFinish(glyph, handles);
   end;
end;

% Return something if required
if (nargout > 0),
   h = hndl;
end;
```

```
%DOUBLE/SURFACE        Create a surface.
%   If the parent axes are shared, then so will the surface. If there is no
%   tag, one will be generated. Otherwise, as SURFACE.

function h = surface(varargin)

% Run the original command
```

```
hndl = builtin('surface',varargin{:});
phndl = builtin('get',hndl,'Parent');
root = getappdata(0,'Rune');

% If the parent axes are a glyph, make the child a glyph
if (isappdata(phndl,'Rune')),
   tag = get(hndl,'tag'); parent = getappdata(phndl,'Rune');
   parent = getappdata(phndl,'Rune');
   if (isempty(tag)),        % generate a name if not given one
      tag = root.freshIdentifier([]); tag = tag.toString;
      builtin('set',hndl,'tag',tag);
   end;
   name = parent.seek(tag);
   location = getappdata(0,'RuneLocation');
   if (~name.isComplete),        % Totally new surface
      keys = lower(fieldnames(builtin('set',hndl)));
      values = unempty(builtin('get',hndl,keys)');
      glyph = parent.budMatlab(tag, hndl, 'surface', keys, values);
      setappdata(hndl, 'Rune',glyph); runify(hndl);
   elseif (name.isFrom(location)),        % Existing surface - FIXME
      builtin('delete',hndl);
      hndl = findobj(0,'tag',tag,'type','surface');
      return;
   else,                         % Grafted surface
      details = parent.graftStart(name);
      details = unmarshal(details);
      handles = runify(hndl, 0, details); glyph = details{5};
      parent.graftFinish(glyph, handles);
   end;
end;

% Return something if required
if (nargout > 0),
   h = hndl;
end;
```

```
%DOUBLE/TEXT        Text annotation.
%   If the parent axes is shared, the text will also be shared. If the text
%   has no tag, one will be provided. Otherwise, as LINE.

function h = text(varargin)

% Run the original command and parse the output
hndls = builtin('text',varargin{:});
phndls = builtin('get',hndls,'Parent');
nlines = length(hndls);
if (nlines > 1), phndls = cat(1,phndls{:}); end;
root = getappdata(0,'Rune');

% For each text label, if the parent is a glyph, make the child a glyph
```

```
for (i = 1:nlines),
   if (isappdata(phndls(i),'Rune')),
      tag = get(hndls(i),'Tag');
      parent = getappdata(phndls(i),'Rune');
      if (isempty(tag)),        % generate a name if not given one
         tag = root.freshIdentifier([]); tag = tag.toString;
         builtin('set',hndls(i),'tag',tag);
      end;
      name = parent.seek(tag);
      location = getappdata(0,'RuneLocation');
      if (~name.isComplete),                          % Totally new text
         keys = lower(fieldnames(builtin('set',hndls(i))));
         values = unempty(builtin('get',hndls(i),keys)');
         glyph = parent.budMatlab(tag, hndls(i), 'text', keys, values);
         setappdata(hndls(i),'Rune',glyph); runify(hndls(i));
      elseif (name.isFrom(location)),        % Existing text - FIXME
         builtin('delete',hndls(i));
         hndls(i) = findobj(0,'tag',tag,'type','text');
         return;
      else,                                          % Grafted text
         details = parent.graftStart(name);
         details = unmarshal(details);
         handles = runify(hndls(i), 0, details);
         glyph = details{5};
         parent.graftFinish(glyph, handles);
      end;
   end;
end;

% Return something if required
if (nargout > 0),
   h = hndls;
end;
```

```
%DOUBLE/UICONTROL        Make user interface controls.
%   If the parent figure is shared and the uicontrol is tagged, it will be
%   shared also. Otherwise, as UICONTROL.

function h = uicontrol(varargin)

% Run the original command
hndl = builtin('uicontrol',varargin{:});
phndl = builtin('get',hndl,'Parent');

% If the parent is a glyph, and the new child is named, make it a glyph
if (isrune(phndl)),
   tag = get(hndl,'Tag');
   if (~isempty(tag)),
```

```
      parent = getappdata(phndl,'Rune');
      name = parent.seek(tag);
      location = getappdata(0,'RuneLocation');
      if (~name.isComplete),                        % Totally new uicontrol
          keys = lower(fieldnames(builtin('set',hndl)));
          values = unempty(builtin('get',hndl,keys)');
          glyph = parent.budMatlab(tag, hndl, 'uicontrol', keys, values);
          setappdata(hndl,'Rune',glyph); runify(hndl);
      elseif (name.isFrom(location)),        % Existing uicontrol
         builtin('delete',hndl);
         hndl = findobj(0,'tag',tag,'type','uicontrol');
         return;
      else,                                          % Grafted uicontrol
         details = parent.graftStart(name);
         details = unmarshal(details);
         handles = runify(hndl, 0, details);
         glyph = details{5};
         parent.graftFinish(glyph, handles);
      end;
   end;
end;

if (nargout > 0),
   h = hndl;
end;
```

```
%DOUBLE/UIMENU           Create user interface menus.
%   If the parent figure is shared and this menu is tagged, it will also be
%   shared. If the parent menu is shared, this menu will also be shared.

function h = uimenu(varargin)

% Run the original command
hndl = builtin('uimenu',varargin{:});
phndl = builtin('get',hndl,'parent');
pkind = builtin('get',phndl,'type');
tag = builtin('get',hndl,'tag');

% Decide whether to runify it
switch (pkind),
   case 'figure',
      runifyit = (isrune(phndl) & ~isempty(tag));
   case 'uimenu',
      runifyit = isrune(phndl);
   otherwise,
      runifyit = 0;        % Que?
end;

if (runifyit),
```

```
   if (isempty(tag)),                % If no tag, grab a fresh one
      root = getappdata(0,'Rune');
      tag = root.freshIdentifier([]); tag = tag.toString;
      builtin('set',hndl,'tag',tag);
   end;
   parent = getappdata(phndl,'Rune');
   name = parent.seek(tag);
   location = getappdata(0,'RuneLocation');
   if (~name.isComplete),        % Totally new uimenu
      keys = lower(fieldnames(builtin('set',hndl)));
      values = unempty(builtin('get',hndl,keys)');
      glyph = parent.budMatlab(tag, hndl, 'uimenu', keys, values);
      setappdata(hndl,'Rune',glyph); runify(hndl);
   elseif (name.isFrom(location)),        % Already here. Sigh.
      builtin('delete',hndl);
      hndl = findobj(0,'tag',tag,'kind','uimenu');
      return;
   else,
      details = parent.graftStart(name);
      details = unmarshal(details);
      handles = runify(hndl, 0, details);
      glyph = details{5};
      parent.graftFinish(glyph, handles);
   end;
end;

if (nargout > 0),
   h = hndl;
end;
```

Matlab callback definitions

The following are meant to be stored as callbacks in shared handle graphics objects. They query the facet for the callback value to execute, and then execute it. How the facet notifies the rest of the mirror is described in section 3.3.3. The full set of callback properties are `Callback`, `ButtonDownFcn`, `CloseReqFcn`, `CreateFcn`, `DeleteFcn`, `KeyPressFcn`, `ResizeFcn`, `WindowButtonDownFcn`, `WindowButtonUpFcn`, `WindowButtonMotionFcn` and `CloseRequestFcn`. Representative samples are listed below; the others are similar.

```
%RUNECALLBACK    Slot-in for the Callback.
function RuneCallback()

h = builtin('get',0,'CallbackObject');
```

```
if (isempty(h)), return; end;

% Retrieve and execute callback.
if (isappdata(h,'RuneCallback')),
   c = getappdata(h,'RuneCallback'); eval(c); return;
else,
   k = lower(builtin('get',h,'style'));
   r = getappdata(h,'Rune');
   c = r.get('callback',1); c = c.toString; eval(c);
   switch (k),
      case {'edit','text'}
         r.set('string',builtin('get',h,'string'));
      otherwise,
         r.set('value',builtin('get',h,'value'));
   end;
end;



%RUNEBUTTONDOWN    Slot-in for the ButtonDownFcn callback.
function RuneButtonDown()

h = builtin('get',0,'CallbackObject');
if (isempty(h)), return; end;
if (isappdata(h,'RuneButtonDownFcn')),
   c = getappdata(h,'RuneButtonDownFcn'); eval(c);
else,
   r = getappdata(h,'Rune');
   c = r.get('buttondownfcn',1); c = c.toString;
   eval(c);
end;



%RUNECREATE    Slot-in for the CreateFcn callback.

function RuneCreate

h = get(0,'CallbackObject');
unrunify(h);




%RUNEDELETE        Slot-in for the DeleteFcn callback.
function RuneDelete()

h = get(0,'CallbackObject');
if (isempty(h)), return; end;

% Get the DeleteFcn
if (isappdata(h,'RuneDeleteFcn')),
```

```
   call = getappdata(h,'RuneDeleteFcn');
else,
   r = getappdata(h,'Rune');
   call = r.get('deletefcn',1); call = call.toString;
end;

% Do the deed
r.dispose;
eval(call);
```

Matlab/Java bridging code

The various interfaces between Matlab and Java are subtly inconsistent. The functions below are used by the rest of the `mirror` toolbox to compensate for this, and for other minor utilities.

```
%UNMANGLE Turn some keys and values into acceptable properties.
%   Feed it a cell array of property names, and another of values.
%   It will spit out the corrected keys and values.
%   Intended for use by DOUBLE/SET.

function [newkeys, newvals] = unmangle(keys,values)

% Ensure the arguments are cell arrays - wasteful, but...
if (ischar(keys)), keys = {keys}; values = {values}; end;

% Unmangle the values into a new pair of cell vectors.
nvals = length(keys); newvals = cell(size(values));
for i=1:nvals,
   switch lower(keys{i}),
      otherwise,
            % N by M matrices are split into cell arrays; need concatenating
         if (iscell(values{i}) & ~isempty(values{i}) ...
             & isa(values{i}{1},'double')),
            newvals{i} = cat(1,values{i}{:})
            disp('Catting.');
         else,
            newvals{i} = values{i};
         end;
   end;
end;

% Weed out properties that can't be transferred.
forbidden = {'currentaxes','currentmenu','currentobject','currentfigure',...
  'parent', 'children','uicontextmenu', ...
  'pointershapehotspot','pointershapecdata', ...
  'xlabel','ylabel','zlabel','title'};
[newkeys, idx] = setdiff(keys,forbidden);
newvals = newvals(idx);
```

```
%UNEMPTY        Replaces [] with {} in cell arrays, sans error checking.
%   Necessary because the Java/Matlab interface can't handle cell arrays
%   with both empty strings '' and empty numeric matrices [] in them.
%   Empty cell arrays {} make a good substitute, which is reversed by
%   whichever Java function uses the cell array.

function out = unempty(in)

if (~iscell(in)), out = in; return; end;

out = cell(size(in));
for (i=1:length(in)),
   x = in{i};
   if (isempty(x)),
      if (isa(x,'double')),
         out{i} = {};
      else
         out{i} = x;
      end;
   else,
      if (iscell(x)),
         out{i} = unempty(x);
      else,
         out{i} = x;
      end;
   end;
end;
```

```
%STDSET           Set object properties.
%   Slow as sin, due to substantial preprocessing.

function success = stdset(hndl, props, values)

try,
   [props, values] = unmangle(props, values);
   builtin('set',hndl,props,values); success = 1;
catch,
   warning(lasterr);
   success = 0;
end;
```

```
%SETARGS Dissects input arguments for DOUBLE/SET and STDSET.
function [props, vals, nkeys] = setargs(varargin)

if (nargin == 0),
   nkeys = 0; props = []; vals = [];
elseif ((nargin==1) & ischar(varargin{1})),
   nkeys = 0; props = []; vals = [];
```

```
else
   nkeys = 0; i = 1;
   props = {}; vals = {};
   while (i <= nargin),
      current = varargin{i};
      if (ischar(current)),        % ...'Property',Value,...
         if (i+1 <= nargin),
            nkeys = nkeys + 1;
            props{nkeys} = current; vals{nkeys} = varargin{i+1};
            i = i+2;
         else,
            error(['Missing property for ' current]);
         end;
      elseif (iscell(current)),               % Cell array of properties, values
         if (i+1 <= nargin),
            values = varargin{i+1};
            if (iscell(values)),
               nkeys = nkeys + length(values);
               props = cat(1, props, current);
               vals = cat(1, vals, current);
               i = i+2;
            else,
               error('Missing property array');
            end;
         else,
            error('Missing property array at end of arguments');
         end;
      elseif (isstruct(current)),        % Structure array of properties, values
         keys = fieldnames(current);
         values = struct2cell(current);
         nkeys = nkeys + length(values);
         props = cat(1, props, keys);
         vals = cat(1, vals, values);
         i = i+1;
      end;
   end;
end;
```

Runification code

The functions below implement the Matlab side of the adoption and grafting process described in section 3.3.3 and in previous appendices.

```
%RUNIFY   Interpret a Concurrent Matlab runification request.
%   Usage: out = runify(hndl,recursive, details)
%   details = {"tag", "kind", {keys}, {values}, glyph, children...}

function out = runify(hndl, recursive, details)
```

```
% Parse details, install sundries as needed
if (nargin == 1),
   recursive = 0; kind = lower(builtin('get',hndl,'type'));
else,
   tag = details{1}; kind = details{2};
   if (recursive),        % Does not exist yet
      parent = hndl; hndl = builtin(kind,'tag',tag,'parent',parent);
   end;
   disp(['Runifying ' num2str(hndl)]);
   stdset(hndl, details{3}', details{4}');
   glyph = details{5};
   setappdata(hndl,'Rune',glyph);
   recursive = 1;
end;


% Install callbacks - susceptible to optimization
stdset(hndl, 'ButtonDownFcn','RuneButtonDown');
stdset(hndl, 'CreateFcn','RuneCreate');
stdset(hndl, 'DeleteFcn','RuneDelete');
switch (kind),
   case 'figure',
      stdset(hndl,'WindowButtonDownFcn','RuneWindowDown');
      stdset(hndl,'WindowButtonUpFcn','RuneWindowUp');
      stdset(hndl,'WindowButtonMotionFcn','RuneWindowMotion');
      stdset(hndl,'CloseRequestFcn','RuneCloseReq');
      stdset(hndl,'KeypressFcn','RunePress');
      stdset(hndl,'ResizeFcn','RuneResize');
   case {'uicontrol','uimenu','uicontextmenu'},
      stdset(hndl,'Callback','RuneCallback');
end;

% Recurse through children
if (recursive),
   nkids = length(details) - 5;
   out = cell(nkids+1,1);
   out{1} = hndl;
   for (i=1:nkids),
      out{i+1} = runify(hndl,1,details{i+5});
   end;
end;




%DOUBLE/UNRUNIFY    Makes a shared HG object unshared.

function unrunify(hndl)


rune = getappdata(hndl,'Rune');
kind = builtin('get',hndl,'type');

% Nuke all children.
```

```
for (i = allchild(hndl)),
   if isappdata(i,'Rune'),
      unrunify(i);
   end;
end;

% Change various properties back.
putback(hndl, rune, {'buttondownfcn','createfcn','deletefcn'});
switch (kind),
   case {'uicontrol', 'uimenu', 'uicontextmenu'},
      putback(hndl, rune, {'callback'});
   case 'figure',
      putback(hndl, rune, {'closerequestfcn','resizefcn','keypressfcn',...
                           'windowbuttondownfcn','windowbuttonupfcn',...
                           'windowbuttonmotionfcn'});
end;

% Erase the other evidence.
rmappdata(hndl,'Rune');
rune.dispose;


%PUTBACK        Put HG properties back the way they should be
function putback(hndl, rune, properties)
values = cells(size(properties));
for (i = 1:length(properties)), values{i} = rune.get(properties{i},0); end;
builtin('set',hndl,properties,values);
```

Appendix C

Test Case

This is the Matlab demonstration application used in chapter 4. The version printed here will work without modification on Matlab over T.128 and Concurrent Matlab over RMI.

```
%TEST        Tiny test case for Concurrent Matlab

function test(role)

global sFig sAxes sLine1 sLine2 sRed sBlue sText
if (nargin < 1), role = 'init'; end;

switch role,
   case 'init',
      sFig = figure('tag','sFig',...
         'position',[400,400,400,400]);
      sAxes = axes('tag','sAxes',...
         'position',[0,0.4,1.0,0.6]);
      x = sort(rand(25,1)); y = sort(rand(25,1));
      sLine1 = line(x,y); sLine2 = line(y,x);
      sRed = uicontrol('style','pushbutton','tag','red',...
         'string','Redraw', 'callback','test red',...
         'backgroundcolor','red','position',[0,80,200,80]);
      sBlue = uicontrol('style','pushbutton','tag','blue',...
         'string','Run', 'callback','test blue',...
         'backgroundcolor','blue','position',[200,80,200,80]);
      sText = uicontrol('style','edit','tag','sText',...
         'HorizontalAlignment','left','min',0,'max',4,...
         'backgroundcolor','white','position',[0,0,400,80]);
   case 'red',
      x = sort(rand(25,1)); y = sort(rand(25,1));
      set(sLine1, 'XData',x,'YData',y);
```

```
      set(sLine2, 'XData',y','YData',x);
   case 'blue',
      t = get(sText,'string');
      out = evalc(t);
      set(sText,'string',out);
end;
```

Bibliography

[1] C. Bajaj and S. Cutchin. Web based collaborative visualization of distributed and parallel simulation. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium*. IEEE, October 1999.

[2] James Begole, Mary Beth Rosson, and Clifford Shaffer. Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer-Human Interaction*, 6(2), June 1999.

[3] Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of the 1994 Supercomputing Symposium*. SC94, June 1994.

[4] C.A.Ellis and S.J.Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM, 1989.

[5] Jim X. Chen, David Rine, and Horst D. Simon. Advancing interactive visualization and computational steering. *IEEE Computational Science and Engineering*, December 1996.

[6] Scott C. Doney. Major challenges confronting marine biogeochemical modelling. *Global Biochemical Cycles*, 13(3), September 1999.

[7] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[8] W3C DOM Working Group. *Document Object Model (DOM) Level 1 Specification (Second Edition)*. World Wide Web Consortium, 2000.

[9] Robert Haimes. pv3: A distributed system for large-scale unsteady CFD visualization. Technical Report 94-0321, American Institute of Aeronautics and Astronautics, 1994.

[10] International Telecommunication Union. *Recommendation T.128 - Multipoint application sharing*, February 1998.

[11] J.F.Baldomero. LAM/MPI toolbox for matlab. World Wide Web, http://atc.ugr.es/javier-bin/mpitb_eng.

[12] Kirk E. Jordan, David A. Yuen, Davis M. Reuteler, Shuxia Zhang, and Robert Haimes. Parallel interactive visualization of 3d mantle convection. *IEEE Computation Science and Engineering*, December 1996.

[13] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[14] Junwei Lu. ACVEM - Applied Computational and Visual Electromagnetics for Computer Aided engineering in Classroom. In *IEEE Conference on Electromagnetic Field Computation*. IEEE, 2000.

[15] The Mathworks. *MATLAB Application Program Interface Guide*, 5.2 edition, 1998.

[16] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1995.

[17] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, 1996.

[18] Ernest H. Page and Jeffrey M. Opper. Observations on the complexity of composable simulation. In *Proceedings of the 1999 Winter Simulation Conference*. MITRE Corporation, 1999.

[19] Alex Pang, Craig M. Wittenbrink, and Tom Goodman. CSpray: A collaborative scientific visualization application. *IEEE Multimedia*, March 1997.

[20] Pau'l Pauca, Kun Liu, Joel Hollingsworth, and Rudy Martinez. Parallel toolbox for matlab. World Wide Web, http://www.mthcsc.wfu.edu/pt/pt.html, 1995.

[21] S. Pawletta, T. Pawletta, and W. Drewelow. Distributed and parallel simulation in an interactive environment. Technical report, University of Rostock, 1995.

[22] P.Husbands and C.Isbell. MITMatlab: A tool for interactive supercomputing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1999.

[23] A. Saran, D. Agrawal, A. El Abbadi, T.R.Smith, and J.Su. Scientific modeling using distributed resources. In *Proceedings of the 1996 Conference on Geographical Information Systems*. Association for Computing Machinery, 1996.

[24] Paul L. Springer. Matpar: Parallel extensions for MATLAB. Technical report, Jet Propulsion Laboratory, 1998.

[25] C. Sun and C.A.Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, pages 59–68. ACM, May 1998.

[26] C. Sun and David Chen. Consistency maintenance and conflict resolution in real-time cooperative graphics editing systems. In *Proceedings of the Inaugural Australian Symposium on Computer-Supported Cooperative Work*, pages 31–37. IEEE, August 1996.

[27] C. Sun and David Chen. A multi-version approach to conflict resolution in distributed groupware systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 316–326. IEEE, April 1999.

[28] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.

[29] C. Sun and Rok Sosic. Optional locking integrated with operational transformation in distributed real-time group editors. In *Proceedings of the ACM 18th Symposium on Principles of Distributed Computing*, pages 43–52. ACM, August 1999.

[30] Anne E. Trefethen, Vijay S. Menon, Chi-Chao Chang, Grzegorz J. Czajkowski, Chris Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on multiple processors. In *SC96 Proceedings*. ACM and IEEE, 1996.

[31] Jason Wood, Helene Wright, and Ken Brodie. Collaborative visualization. Technical report, University of Leeds, 1996.

[32] John A. Zollweg and Arun Verma. *Cornell Multitask Toolbox for MATLAB*. Cornell Theory Centre, 2000.