Dreme: for Life in the Net

by

Matthew Fuchs

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Computer Science

New York University

September, 1995

Approved

In Memory of Harry and Sylvia Landes

but also for

Annie, Yael, and Elie

All treasures beyond reckoning

Acknowledgments

At NYU, special mention must go to David Fox for pointing out that NYU wants a mixture of cotton, dead trees, and fine black dust, not magnetized mylar. Otherwise I'd have probably never taken the time out from programming to produce the current *pavé*. David also deserves mention for being a great officemate and having eclectic tastes in music (and a stereo in the office).

I want to thank Ken for leaving me alone when he could have insisted I Do Something as Required. Now that this is finally over, I hope we'll be able to work together, and he'll see the fruits of his investment. I've been remarkably lucky in finding bosses who leave me alone, and I hope they won't all be disappointed.

David Bacon and Marco Antonietti also deserve mention for disagreeing relentlessly. Fortunately that required physical displacement. I also want to thank Gary and the crew at the Courant Library. I enjoyed talking with them immensely.

At CERC, I want to especially thank Ramana Reddy and V. J. Jagannathan for an inordinate amount of faith – 20 minutes on the phone and no letters of recommendation. Much of what you will soon read about was perfected at CERC, where I have been given wonderful support for the past year.

In the background lie David and David of cfX, who paid me good money to do what I wanted, on their time. I hope they derived as much benefit as I did. Alex Mogielnicki made life infinitely easier as I prepared for grad school. Alex claimed he was repaying those who enable him to get his doctorate. Makes me anxious to have an employee so I can pay him back. And, of course, there's Ken Garvey and Bill Meckel.

My parents have given me unrelenting support for longer than I can remember, for obvious reasons. This is my first chance to thank them publicly, in writing.

Recounting the reasons this would not have been possible without them would require another tome. I love them.

Harry Landes, my maternal grandfather, didn't live to see me enter graduate school. I regret not having had the opportunity to attempt to explain to him what I was doing. I can only hope he'd have been pleased.

Finally, there are the important people. *Chère* Annie has put up with innumerable explanations of theory, networks, SGML, etc., not to mention my typing through most of many nights, with remarkable humor. France, fortunately, has a tradition of supporting its intellectuals. Yael and Elie probably don't realize that there are parents who aren't working on dissertations, but they'll find out soon.

Title: Dreme: for Life in the Net

Author: Matthew Fuchs

Advisor: Prof. Ken Perlin

Dissertation Abstract

This dissertation makes four contributions towards supporting distributed, multi-user applications over open networks.

Dreme, a distributed dialect of the Scheme language in which all first-class language objects are mobile in the network. In particular, various distributed topologies, such as client/server and peer-to-peer, can be created by migrating closures with overlapping scopes around the network, correct inter-process communication being assured by Scheme's lexical scoping rules and network wide addressing. Threads of control are passed around through first-class distributed continuations.

A User Interface toolkit for coordinating events in multi-threaded, multi-user applications by organizing continuation callbacks into nested lexical scopes. Each event has certain attributes, such as synchronous/asynchronous. Certain events create new scopes with new events. Continuation callbacks allow both synchronous events which return values to their callers, and asynchronous ones. Application needn't be spread throughout the application, as with applications using an event-loop.

A distributed garbage collection algorithm that collects all cycles on an open network. The basic algorithm depends on maintaining the inverse reference graph (IRG) among network nodes (i.e., if a-¿b is in the regular graph, b-¿a is in the IRG). A single IRG traversal from any object determines the status of each object touched. Communication is decentralized (any object can choose to determine its status), garbage is touched O(1) times (in the absence of failures), it is fault-tolerant, and can handle malicious or faulty neighbors. Each operation uses messages linear in the size of the IRG. Overlapping operations perform like parallel quick sort.

An approach to using the Standard Generalized Markup Language (SGML) over the network to support distributed GUIs, intelligent clients, and mobile agents. SGML is a meta-grammar for creating domain specific document markup languages to which a variety of semantics (display, reading/writing databases, etc.) can be applied. The document, its grammar, and some semantics, are retrieved over the network. Applications normally create interfaces directly out of graphic objects to communicate with the user. However, if the interface has some semantics (and is parsable), a computational agent can interpret the interface and talk directly to the application on behalf of the human.

Chapter 1

Introduction

In the early days of computing, computers stood alone. Today, computers are linked in ever greater numbers, but processes are still seen as standing alone, or gingerly communicating with one or two "servers" or remote procedures. Current parallel or distributed languages generate any number of communicating elements, but these are usually self-contained, communicating only with each other, as if they were colonizing the virgin territory of an empty network. When finished, they disappear. But most of the networks of the future will not be empty. They will consist of lots of individual inhabitants, humans or otherwise, who wish to communicate with each other. This environment will be very different from both the splendid isolation of the single threaded past and the geometric purity expected by functional languages. Some expect it to be a well run military unit. But, in truth, it will be more like lunchtime in midtown Manhattan!

We postulate a world divided into many users with their own discrete and private domains. Distributed around a network of unknown size and topology, each user is both a provider and consumer of information and services. Users will typically need to browse around the network, locate some group of entities and then orchestrate their cooperation to fulfill some user goal. This environment is implicit in the current debate about national high-speed data networks, interactive

TV, and the predicted spread of personal communications systems and "information appliances". Part of the challenge of creating this world is strictly engineering in the classical sense – creating the high-speed network infrastructure the communication will travel on. Another part is producing the software tools that will facilitate building the distributed, multi-user applications that will run on this network and exploit its resources.

This dissertation attacks the latter part of this challenge by describing three specific elements – a language, a garbage collection algorithm, and a graphical user interace (GUI), all distributed – which together greatly ease the production of networked applications. The structure of these elements is based on four ideas and their implications:

1. Networks should only speak when spoken to and otherwise be invisible. As we shall see, this implies that objects in the network (i.e., code and data) can move through the network as necessary. Nevertheless, facilities should be available for applications to query and manipulate the network when necessary.

2. If we elide the network, as suggested in the previous point, then client/server computing is just an example of using higher order functions.

3. Garbage collection will still be necessary in large networks. However, when the network is too large for centralized control of the process, responsibility falls on the object to manage its own collection.

4. A graphical interface is necessary to orchestrate numerous distributed objects. Current human-computer interface technology is insufficient because it does not adequately support the multi-threaded nature of user-interfaces.

By developing the appropriate support at all levels of the interface, we can correct the deficiencies.

We will examine each of these ideas in turn, show how they relate to each other, and then discuss Dreme, a system built on these ideas which includes the elements mentioned above, and which is the subject of the rest of this work. Security issues are not addressed exhaustively by the present work, but extensions to support secure computation are mentioned where relevent, and a longer discussion will be provided in the conclusion.

1.0.1 Speak when spoken too

Ever since computers started to be connected, there has been a continuous effort to remove this connection from the programmer's list of concerns. Manifestations of this effort include sockets, RPC protocols, and the ISO/OSI layered network model, not to mention the development of the client/server model for distributed computing. We could also add automatic parallelization and coordination languages, such as Linda, to this list. All attempt to reduce the importance of location as a significant factor in developing computer systems. Following in the line of Emerald[50, 28], we propose to go farther by allowing application objects to move around the network during the course of computation.

With object mobility, we can support the same basic software components in a changing environment in a fashion that is much more difficult without mobility. Among these benefits are:

1. On-demand delivery of new software components. In a large distributed network, people will need to communicate with services, or participate in distributed applications, for which they do not have the requisite software.

The necessary components can be delivered to them as they are required. This extends from application code to instantiated objects and threads of control.

2. More flexible use of network resources. Part of a computation can be moved from a workstation to a local supercomputer or back again, depending on the deadline for the results. Applications running on, for example, hand-held devices with limited memory can off load parts of the computation to more powerful machines elsewhere on the network.

3. Fine-grained load balancing, as any element of a computation is potentially movable over the course of a computation (although the cost of moving a computation must be balanced against the cost of performing it). This implicitly assumes a two-layer architecture with a layer of application objects and a location manager which dynamically changes the location of these objects.

4. Greater security control. In the face of possible malicious intent, it must be considered that any information arriving at a particular node is freely accessible, despite any language constructs enforcing encapsulation. While mobility allows better exploitation of the client's resources, selective *immobility* can keep information off of untrusted systems and allow it to be moved to trusted ones, without changing the application.

There are times when it is important to talk to the network, particularly in a heterogenous network where resources are not evently distributed. For example, multi-user applications may need to locate their users (and vice-versa), clients need to locate servers, and non-replicated data must be tracked to its source. Nevertheless, conversing with the network should be at as high a level as possible. Separate processes, with their attendant address spaces, are just as much of an impediment as the hardware network. To completely expunge the network from consideration, even processes must be seen as no more than containers for the objects actually performing computation.

Their are two fundamental categories of interaction that an application must have with the network.

1. *Find an object.* Every time an application wishes to communicate with an object, that object must be found; there is no guarantee the object is in exactly the same place as it was for the last communication. We can distinguish, however, between locating an object for the first time, an arbitrarily complicated process possibly involving database queries, etc., and keeping track of that object. We can assist in the first process by placing the network in a unified name space and providing a low level uniform reference mechanism to mask the topology of the network. We can support the second by having the run-time system keep track of object movements, relieving applications of this task.

2. *Place objects around the network.* Different nodes of a network have different features. To take advantage of this, applications may need to ship objects to other nodes for special handling. They may also need to start applications on the target node if they are not already present. If we push this to the limit, then our (possibly unrealizable) goal should be that the processing node in the network on which an instruction is executed may be arbitrarily chosen at runtime.

Several researchers have dealt with the issue of uniform distributed addressing[7, 54, 6, 1]; we shall present our variation on this, which provides some asymptotic improvements, when discussing garbage collection.

Placing an object, and in particular, placing an instruction, on a node, is a more complicated task. Given a large heterogenous network with ad hoc communication patterns, we must be able to move not just data, but the executable code as well; if the recipient has it already, that is fine, but we cannot assume that is generally true. To make this mobility useful, the code and data need to be ready to perform as soon as they arrive at the receiver, otherwise the network intrudes again (the data transmission rate is an unavoidable exception to network transparency). The more traditional means of moving code around the Internet, ftp and the postal service, require too much user intervention. In the current work we will mostly consider explicit mobility, although in the larger sense, we must consider automatic parallelization, and we will allude to that later.

If we wish to communicate with end users, location transparency also means that an object arriving on a particular node must be able to display its interface to a user at that location. As this should be a graphical interface, we must handle one aspect of the heterogeneity of the network, the variety of GUI systems, cleanly, so that an object needs only one interface which can be displayed on any node. Our approach to this problem will be discussed in the chapter on user interfaces.

1.0.2 Servers and higher-order functions

Client/Server is an oft used term whose meaning seems to be as follows:

A Client and a Server are two processes, or two objects, which have a pattern of communication in which the Client makes requests of the Server and the Server fulfills them, independent of the locations of the Client and Server in the network.

In practice, Client/Server is mostly used to refer to systems where each part resides in a separate process, or even a separate network node. Frequently there is an attempt to make the Server appear to the Client as just another function call through a remote procedure call mechanism, although Servers may service several Clients at the same time, so the relationship is not symmetric. If we accept that the network be quasi-invisible, though, we can arbitrarily place the Client and Server in the same or different processes – the communication should appear the same whether they are in consecutive bytes in memory or on different continents.

If we examine the Client/Server interaction more closely, we see that the Server, wherever it is, exists to provide access to information or services which the Client, removed from that information, cannot directly manipulate. From the Client's perspective, then, the Server is a preexisting function providing an interface to some amount of encapsulated information. The Client sends a request or data, and the Server manipulates it based on this encapsulated information.

A Server, then, is fundamentally a closure of the type found in high-level languages such as ML, Scheme, or Haskell, except that Servers are usually built in an environment (C, C++, Cobol) which does not have an easy representation for higher-order functions and where significant work needs to be done to overcome process/network boundaries. If we have a language which has higher-order functions and can unify local and remote communication, then Servers are ordinary higher-order functions which don't require special support except as they have special requirements (eg., multi-threading and security control). A higher-order function can be called to generate a Server. The Server becomes distributed when its location is known on another node. The network should play little or no explicit role; concurrency adds sufficient complexity on its own.

The Client in a typical application is either application code itself that talks directly to the Server through some form of IPC, or may consist of special Client code linked into the local application. This Client code knows how to communicate with the Server, and serves as an intermediary between it and the application. In this case, the Client plays the same role *vis-a-vis* the application that the Server does *vis-a-vis* it, except that here, the Client encapsulates special information about the structure of the server. This private shared information can be thought of as an overlapping environment.

We can obtain this situation quite easily if the Server, like its hypothetical creator, also returns a function when called. In this case, the Server returns the Client. Again, if we ignore the network, then the communication between the Client and Server can take place through shared functions and variables. Server and Client can also behave as coroutines, such as when a Client sends a request and takes the responses one at a time.

If we generalize this, and combine it with the mobility discussed in the last section, we can evolve from present Client/Server systems, where the choices available are the run-time binding of RPC stubs, to a very fluid one. Users can communicate with any Server by receiving an appropriate Client. By providing a standard interface to the local GUI, as described later, the Client can communicate with the user, and if it provides a known programmatic interface, then it can also communicate with other local objects. Current technology, which makes the distribution of Clients difficult, naturally tends to produce large Clients and Servers. Once they are free to move around the network, Clients and Servers can be, like any other closures, of any size. In fact, the technology would now favor smaller Clients, as they take less time to transmit.

Mobility also means we can distribute the components of Clients and Servers around the network and yet still have each appear as a single object.

1.0.3 Garbage Far and Wide

Once applications start to dynamically allocate their own data, they need to start managing it. This are two general approaches:

1. Memory is explicitly deallocated when the programmer can be sure that it is no longer needed.

2. The available memory is periodically searched to find unreferenced objects, and their memory is then recycled.

The first method is used in real-time applications, where all operations must have bounded durations, or where the environment has no automatic means of scavenging memory. The second method, Garbage Collection (GC), is used by functional-style languages. The kind of distributed system described so far really favors GC over explicit deallocation. Explicit deallocation requires an *a priori* understanding of memory access patterns to determine exactly when memory becomes garbage. This can be extended to distributed systems as long as the distributed access patterns are clear, but in the situation we have described they are not. While some objects may be explicitly deallocated, others may be referenced by remote objects about which the application designer knows nothing.

As will be discussed in greater detail in the chapter on GC, distributed strategies divide along two lines:

1. *Reference Counting*, where each object keeps track of how many other objects refer to it. An object which receives a new reference must explicitly tell the referenced object to increment *reference count* and then tells it to decrement the count when it "forgets" the reference. When an objects reference count

reaches 0, the object is automatically recycled. Although there are some timing considerations to implementing distributed reference counting, the major drawback of reference counting is its inability to deal with cycles.

2. *Mark and Sweep*, where the system periodically sweeps through all memory, determines which objects are garbage, and recycles them. The system starts with a *root-set* of objects which are known to be *live* (i.e., not garbage), marks their transitive closure as also alive, and deletes everything else as garbage (since they cannot be reached from some other live object).

Despite it's inability to handle cycles, reference counting is well suited for distributed systems because the coordination overhead is very low; GC related communication is strictly between between the objects concerned. Each object is responsible for itself, even at the local level.

Current algorithms that collect cycles, distribute or otherwise, are universally of the mark-and-sweep variety. The distributed ones are all divided into local and distributed phases and require significant coordination both at the local and distributed levels. In essence, all the processes which contain references, directly or indirectly, to an object must agree that it is garbage before it can be collected. Responsibility for the object becomes a matter of distributed consensus.

In general, cycles cannot be avoided, but the burden imposed by mark-and-sweep is equally undesirable. The solution we propose is a new distributed GC algorithm which eventually collects all cycles, distributed or otherwise, but which has reference counting's desirable property of making each object responsible for its own collection, thereby eliminating the coordination overhead of mark-and-sweep algorithms. The actual burden, in number of messages, of this algorithm is dependent on the frequency with which distributed collections are attempted, and it is shown how to keep this within reasonable bounds.

1.1 How the Event Loop alters a program

Developing a good user interface is a hard problem involving social/psychological issues as well as technical ones. Meyers (1993) provides a list of these difficulties. Beyond the human-computer interface and graphics issues, however, stands the computer-programmer interface — graphical user interface (GUI) toolkits use a reactive, or event-driven, style that is difficult for programmers, but which seems necessary to handle the ever increasing complexity of the user interface. The problem is even more pronounced in distributed applications, where there are several event-loops to contend with, as well as the thread of control of the application.

We will now take a careful look at reactive programming (and at its underlying architectural feature, the event-loop) to determine if this is essential for a flexible user interface. The analysis will indicate that onerous aspects of the reactive style are more artifacts of the software technology being applied than they are inherent to the problem. We will use the results of this analysis later to propose an alternative architecture that is easier for the programmer without limiting, in any way, the complexity of the user interface. On the contrary, it is easily expandable to distributed applications with multiple threads of control. This is done by returning to the run-time system many decisions which are currently handled by the programmer. The paper concludes with a survey of other attempts to deal with this issue.

1.2 How the event loop alters a program

The *event-loop* is an essential characteristic of modern Graphical User Interface programming. A program sits inside a tight inner loop which continuously gets an event, dispatches it to the appropriate handler, and then returns to waiting for the next event. The event-loop is (and, as we will see, must be) memoryless. Any state needed across events must be maintained explicitly through the handlers. The kind of information (variables and return addresses) that the run-time system automatically maintains for a process normally sits on the stack; this is gone after each event. Loss of this information has serious consequences for how programs are written and is a major reason for the difficulty of GUI programming.

1.2.1 A reactive example

We will start with a somewhat complicated example – the card game, bridge. Bridge is played by four persons seated around a table with a standard deck of fifty-two cards. Opposite persons are partners. We will concentrate on the *rubber*, which is composed of two or more *games* (one set of partners must win two games to win the rubber). A game has four parts:

1. All 52 cards are dealt to the players. Each player has a hand of 13 cards, which will be played in 13 rounds, called *tricks*.

2. The players *bid*. In a round-robin fashion, the players make claims about how many rounds of play they can win. This goes on until three players *pass*. The highest bidder, called the *declarer*, establishes a *contract*. At the end of the game, the score will depend on the declarer's success.

3. All 13 tricks are played. We won't go into all the rules here.

4. The play is scored.

Figure 1.1 shows the control flow of a rubber (leaving out the rules) in pseudo-code. As a serial process, a rubber is a series of nested loops, the outermost for each game, the innermost for each trick. We will later consider a parallel version, where any number of tricks can be played simultaneously.

Suppose we have four players sitting in front of their screens with a complex GUI that lets them consult help facilities, previous games, etc., while playing bridge. Each sees approximately the same screen, only the actions they can take at a particular time is different. After the cards are dealt, it is impossible to program anything approximating this simple control flow with an event-loop.

Concentrating just on the play of a single game (i.e., disregarding other GUI elements), in an event-loop implementation, only the callbacks enabling a single player to bid must be active at the start. That player then clicks on a button indicating his bid. The callback must place the bid in some globally accessible place, and disable itself. It must then check if this is the last bid. If so, it must determine who won the bidding, set things up for the game. If not, then it must enable the next bidder to bid. Finally, it must return to the event-loop. After bidding is finished, a similar process occurs for each round of play. Each callback must determine where in the play it is and set up conditions for the next player's actions.

In other words, in the reactive style, the thread of control is spread out among the callbacks. Each callback must ascertain the current state of the computation, react appropriately, and prepare for the next state. This adds an extra degree of complexity; the straightforward control flow of fig. 1.1 must be systematically transformed. In an application where there is little control flow, such as a drawing

```
procedure rubber(players playerArray[4]) returns integer
{
array gamesWon[2] of integer;
do
   contract := 0;
   declarer := 0;
   leader := 0;
   array game[13][4] of card;

   for player := 0 to 3 do
      playerArray[player].deal(cards);
   od

   do
     get the next player´s bid
     if the bid is higher than contract then
        contract := the new bid
        declarer := the current bidder
        leader := the current bidder
     endif
   until three passes in a row

   for trick := 0 to 12 do
      for player := 0 to 3 do
         game[trick][player] :=
               playerArray[(player + leader) mod 4].playCard();
      od
      leader := (scoreTrick(game[trick]) + leader) mod 4;
   od
   gamesWon[scoreGame(game, declarer)]++;
until (gamesWon[0] = 2 or gamesWon[1] = 2);
return if (gamesWon[0] = 2) then 0 else 1 endif;
}
```

Figure 1.1: Rubber Control Flow

program or a chess game, this is not particularly onerous; where there is significant control flow, as in bridge, it is more so.

The usual justification given for this transformation is user freedom. Graphical programs are user driven, the user should be given the widest possible latitude, the traditional programming style (in which input is all synchronous) constrains this freedom, therefore it must be abandoned. In the current case, the control flow follows the playing of a hand. Without the event loop, if a player wishes to do some other action, involving menus, help, or whatever, they are blocked from doing this; all input is captured by the routines for bidding or playing cards. A more radical example would be altering bridge to allow multiple rubbers to be played simultaneously. Although maximizing the user's freedom of action (within the semantics of the program) is important, we will show that the solution most widely used, the event-loop, is actually a side effect of the technology chosen for implementation.

1.2.2 The essence of the event loop

In the old days when graphical systems were rare and application user interfaces were variations on the menu, the choices open to a user at any point in an application were quite limited and could be handled in a case statement. When the program needed more information from the user, it could call a function to prompt the user. The application could be structured as a set of hierarchical menus accessed through nested function calls, relying on the language to keep track of state (applications written in languages which allowed nested functions, such as Pascal or Ada, could also use scoping to help communicate information from one level to another). This made writing a user interface very straightforward.

GUIs are inherently multithreaded leading to the event-loop. Suppose that there

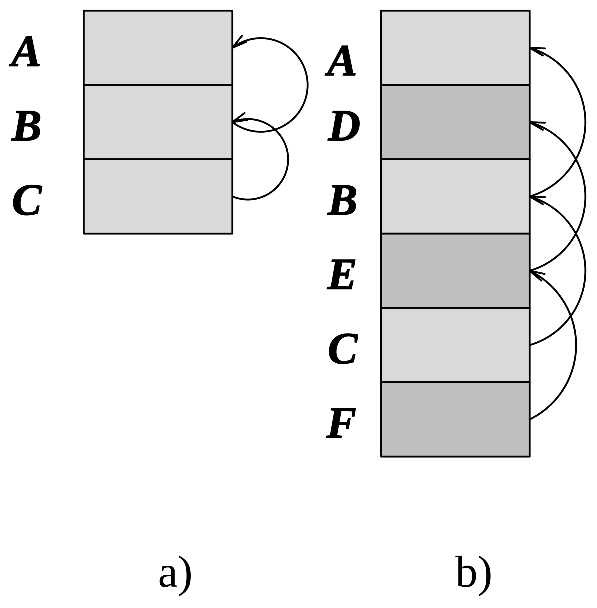


Figure 1.2: Stacks and Threads

is a path through the interface for which history is important, i.e., completing form $\mathcal{A}$ requires the results of form $\mathcal{B}$, which in turn requires the results of form $\mathcal{C}$. $\mathcal{C}$, then, needs to know how to return to $\mathcal{B}$, which needs to know how to return to $\mathcal{A}$. One solution, along the lines used by the old-style menus, would be for the handler that creates form $\mathcal{A}$ to recursively call the handler that creates $\mathcal{B}$ when necessary. $\mathcal{B}$'s handler then likewise creates $\mathcal{C}$. When $\mathcal{C}$ terminates, it hands its results to $\mathcal{B}$, etc. (See figure 1.2a.) However, if we have another group of history-sensitive interface objects ($\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$), then we have a problem. In our Bridge example, a novice user might invoke an interactive tutor facility while choosing a bid. Since the user is free to develop both of these histories in parallel (working consecutively in $\mathcal{A}$, $\mathcal{D}$, $\mathcal{B}$, $\mathcal{E}$, $\mathcal{C}$ and then $\mathcal{F}$), $\mathcal{A}$ would have to call the handler for $\mathcal{D}$, $\mathcal{D}$ the handler for $\mathcal{B}$, etc., and the stack will have the two sequences interleaved. As shown in figure 1.2b, $\mathcal{C}$ is inaccessible until $\mathcal{F}$ completes. In essence, it is impossible to use the stack for control in a modern GUI, and so the stack has been abandoned. The event-loop is a control regime which doesn't require a stack.

With the event-loop, for $\mathcal{A}$ to communicate with $\mathcal{B}$, $\mathcal{A}$ needs to be divided into two parts, $\mathcal{A}$ and $\mathcal{A}$', one doing the work before the call to $\mathcal{B}$ and the other doing the work afterwards. To "call" $\mathcal{B}$, $\mathcal{A}$ calls a separate function β passing a pointer to $\mathcal{A}$' and whatever data is associated with the current invocation of $\mathcal{A}$ (in C++, this is an object and a method). β stores this information, creates the GUI for $\mathcal{B}$, and returns to $\mathcal{A}$ which then returns to the event-loop. When $\mathcal{B}$ terminates (or actually $\mathcal{B}$', since $\mathcal{B}$ must also be split in two pieces) it calls $\mathcal{A}$', passing in the data. $\mathcal{A}$' does a little work (updating fields, activating/deactivating buttons, etc.) and then returns to $\mathcal{B}$' to get back to the event-loop. As a result, for example, it is impossible for a handler to call another piece of interface as if it were a function; the handler must first return to the event-loop for the next event to be retrieved.

The event-loop forces the application programmer to explicitly maintain all the information which in other applications is implicitly maintained on the stack, such as the values of variables and the return location after the next event. The resulting program is, in fact, written in *continuation passing style* (CPS) (Appel 1992). Compilers for functional-style languages, such as Scheme and ML, frequently transform source code into CPS as an intermediate step. Humans normally don't program directly in CPS; it can be quite complex.

In CPS, each function in a program takes an extra parameter, called the *continuation*. Functions never return to their caller. When they have done their piece of work, they call this extra continuation parameter, passing in the result. Since no function ever returns, but simply calls another, a program in CPS, like an event-loop program, cannot use a stack. The GUI programmer programs in this style, but with extra complications (a callback cannot directly call its successor).

When source code is transformed to CPS, the process starts from the outermost functions and works its way in, eventually transforming the body of each source code function into a group of little *continuation functions* which call each other in sequence after performing a small amount of work. Although a GUI programmer has not explicitly performed any such transformation, $\mathcal{A}$' and $\mathcal{B}$' in the previous example are continuation functions organized around user events. By creating several callbacks and then returning to the event-loop, the event-driven program allows the user to "non-deterministically" choose which of several continuations to jump to. The objects created by the programmer in an object-oriented GUI are a replacement for the stack frame; the members are the local variables and the continuation.

Our goal is to provide a "best of both worlds" approach: on the one hand, to allow control flow and other dependencies among events to be programmed in a straightforward fashion, on the other to support the freedom for the user provided by the CPS approach of the event-loop. Given the tension between the two, it is also desireable that it be easy to specify and change these relationships. One resolution to this dilemma, which has been tried (Pike, 1988), would be the wholesale us of explicit threads. We will explore an equivalent, but more flexible, approach here through the use of first-class continuations. The essential insight is, if we use a continuation for a callback, then when the desired event occurs, the program will continue with the next statement, as if it had just been blocked for input. As we shall see, there is still plenty of work to make a usable system from this; the range of behaviors that a user interface needs to support is quite complex.

1.2.3 First-class continuations and callbacks

A first-class continuation is an object that represents the remainder of a computation (Dybvig 1987, Reynolds 1993, Wand 1980). A more concrete description would be an object containing the process stack, α, and the return address, β, just

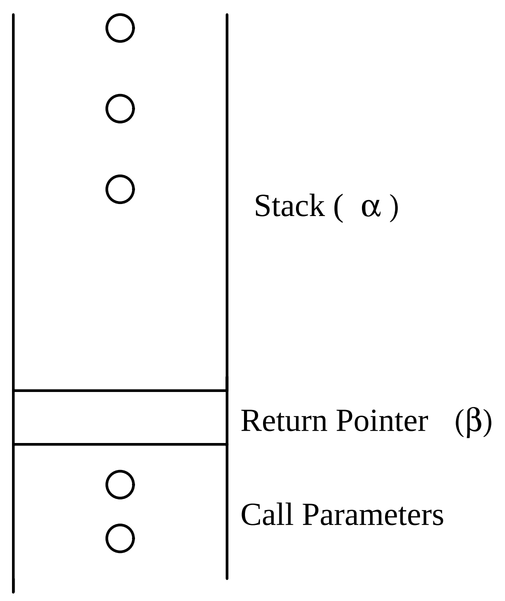


Figure 1.3: Continuations and stack frames

before execution of a function call (fig. 1.3). This continuation acts as a one parameter function; when called with a parameter, γ, it overwrites the current process stack with α, places γ in the return register (or appropriate position in the bottom stack-frame) and jumps to β. In other words, whenever the continuation is called, it is as if the original function call had just terminated, returning the continuation's parameter as answer. The closest C/Unix analog are the functions `setjmp` and `longjmp`, where `setjmp` stores the current values of the registers, including the stack pointer, in a structure, and `longjmp` restores them, peeling the stack back to the point at which `setjmp` was called, as if returning from the initial `setjmp` call. Unlike `setjmp/longjmp`, a first-class continuation can be called at any point in the future computation, repeatedly if desired. (Continuations do not necessarily require copying the stack. Compilers or interpreters using CPS, such as SML/NJ (Appel 1992), do not employ a stack.) The analogy between continuations and `setjmp/longjmp` would be an equivalence if `setjmp` stored the stack as well as the registers.

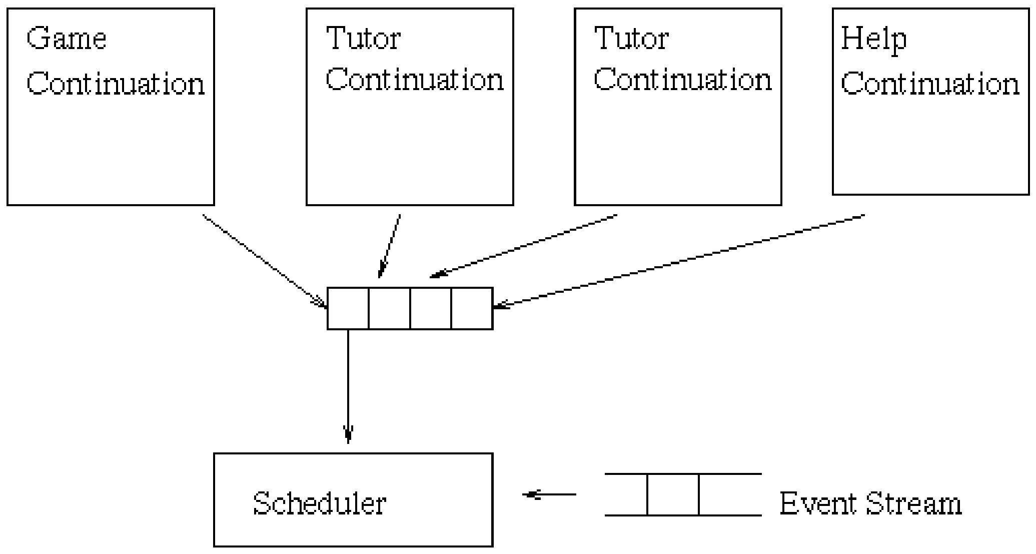


Figure 1.4: Juggling Continuations

Suppose, at the start of a game, a user is engaged in a dialog with a bridge tutor function. When her turn to bid arrives, she starts another dialog with the tutor, and also calls up the help facility (where the help concerns the mechanics of the interface, not the game), before choosing a bid. A number of threads are now mixed on the stack, as in 1.2. While the user ponders, each of these is waiting for input. In other words, each has called some function, *beta*, to retrieve input from the user. Normally the input would go to some callback. Let *beta* instead first take the current continuation of the caller, and store it in a list of pairs, mapping from input events to continuations (fig. 1.4). Now, when the user even occurs, it is passed to the appropriate continuation, restarting that thread. Following the user's actions, *beta* acts as a scheduler (without a timer interrupt, threads only give up control when they require input). Because continuations can be called repeatedly, a dialog with the tutor can be started more than once. Bidding must be more strictly controlled to prevent multiple bids.

Continuation callbacks allow a traditional programming style to be combined with the multiplicity of events in a GUI. Despite the presence of several concurrent threads of control, continuations have allowed us to avoid explicitly creating threads. The next step is further simplify this by hiding the continuations and combining the flow of control of the application with the appearance and disappearance of graphical objects.

We will later present the full graphical user interface, which not only supports a high degree of concurrency, but is also designed to be platform independent, so that objects can display their user-interfaces regardless of the underlying GUI.

1.2.4 Dreme

In order to explore the implications of these ideas, we have built Dreme, a language having a very high level view of the network, an Internet-wide addressing mechanism, garbage collection, object mobility, first-class higher-order functions and first-class continuations. Rather than building this language from scratch, however, we have chosen to modify Scheme, a language which already has garbage collection, first-class higher-order functions and continuations, but no particular policy regarding distribution and interfacing with a network. We have attempted to make minimum alterations to Scheme to support our experimentation. If fact, little was needed, as a single Scheme process on a single machine already has three of the elements we have identified as important, so much of our work has been to cleanly project them onto the distributed world. Conversely, Dreme projected onto a single process is regular Scheme.

One of the goals of Dreme is to make it possible to write distributed applications with little concern for network or process boundaries. Instead, the only boundaries of interest should be those defined by the applications themselves. In Dreme, as in Scheme, boundaries are maintained by closures. Once an application is developed, however, the network inevitably intrudes, whether for reasons of performance, security, or access to particular resources. To compensate for this, Dreme includes several commands as hints to the run time system.

Along with this, we have developed a platform-independent, distributed GUI in Dreme. By platform-independent we mean that a mobile object can display the same interface on a variety of platforms without concern for the host windowing systems. By distributed we mean that parts of an interface can appear in different parts of the network, and that callbacks likewise can traverse the network. As alluded to above, this GUI takes advantage of continuations to eliminate recourse to an event loop.

Chapter 2

Dreme: the distributed extensions

As stated in the Introduction, Dreme is a version of Scheme developed for distributed applications on local area and wide area networks. Excluding these distributed additions, Dreme acts essentially as an ordinary Scheme interpreter compliant with the *Revised*^{4} *Report on the Algorithmic Language Scheme* (R4RS)[12]. This chapter will describe the distributed extensions that make up Dreme.

This chapter consists of seven sections. The first introduces the new language objects introduced by Dreme. We will then examine, in turn mobility, remote invocation, remote evaluation, concurrency, related work and finish with a programming example.

2.1 Dreme language extensions

Given the goals of Dreme, the most urgent matter is deciding how to present the network, and then how to refer to objects as they move around in it. The choice is also somewhat constrained by the choice of the Internet as our target. Methods which might work for shared memory multiprocessors or tightly coupled distributed memory machines are not amenable.

As Scheme itself has no policy towards distribution, Dreme needs a small set of

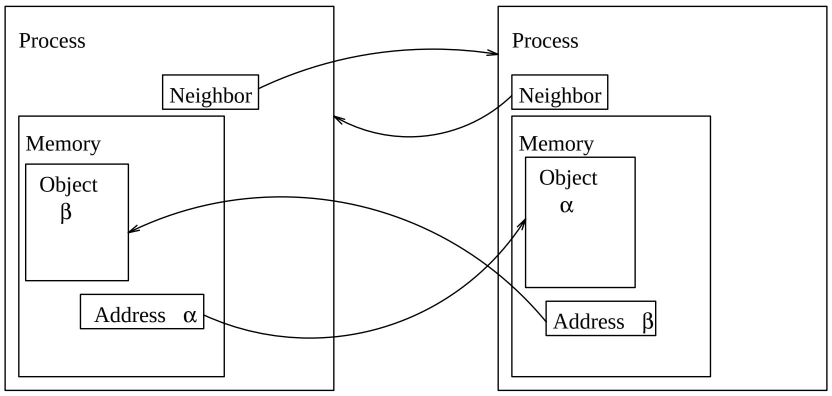


Figure 2.1: Relationships between new object types

new language objects. These are Processes, Neighbors, Addresses, and Memories. The first three are related to distribution, the last to persistence. They let the code access the physical topology of the network and manipulate the location of objects. The relationship between these components is shown schematically in figure 2.1. There are any number of Processes which contain a number of Memories and Neighbors, where each Memory contains objects and Addresses. Each Neighbor points to another Process and each Address points to an object in another Memory.

As will be seen, the behavior of these objects goes beyond just assuring communication among the parts of a distributed Dreme system. They are designed to allow a node to establish policies for the execution environment (if any) of migratory objects. The policies a node chooses with respect to other nodes will vary widely depending on circumstance. For example, a node exposed on the network handling commercial traffic will have very different policies than one executing in a closed environment on a workstation cluster. The former might institute elaborate security policies to protect itself, whereas the latter might institute practically none, as only approved objects can arrive. This chapter does not discuss the elements of these policy decisions, only the mechanisms being provided.

2.1.1 Addresses and Forward References

Addresses are used to coordinate object communication and movement across network nodes. Fortunately, the Internet itself endeavors to present itself to its users at a fairly high level of abstraction. Although divided into several different networks, the whole Internet can be seen as organized into a single hierchical address space of nodes, each of which has a linear set of ports where inter-process communication (IPC) actually takes place. For the assistance of its human users, the Internet also provides a map from strings to nodes. The ports, however, remain numbers, although some ports have been assigned to standard applications. Our naming scheme parallels that of the Internet. Each Address is a string that uniquely identifies a Dreme object in the network. At present, the string associated with the object is its place of origin (Internet node name, Process name, and Memory name), as well as a unique integer (currently 32 bits). Certain objects, such as the Process object, the default Memory, and the default functions, have distinguished Addresses, as they must always exist.

Although all objects implicitly have Addresses, in actuality the Address is stored in a structure that is larger than some application objects. Since each O/S process has its own address space which objects can use to identify each other within a Process, network addressing is only relevant when an object needs to be refered to from another Process in the network. Objects residing in the default Memory and not directly visible beyond a Process boundary do not have Addresses, and must be content with the Unix process address space.

The Address string uniquely identifies an object, but it only specifies its place of origin, and not its current location (otherwise an object's identity would change each time it moves). Addresses can propagate around the network, as described below. To locate mobile objects, each Address comes with a Forward Reference structure indicating the last known location of the object. The fields of the Address object discussed in this chapter are listed in figure 2.2. Address information related to garbage collection will be discussed in the chapter on garbage collection.

To summarize, an object is in one of three stages:

1. If the object is only visible to other object in the local host, then the host process address space is sufficient to identify the object.

2. If a reference to the object crosses boundaries, then the object will need a network Address to identify it.

3. If an object moves while there is an outstanding reference pointing to its current location, then run-time system must also keep track of its new location to route messages.

In the third case, it is only necessary to keep track of an object's location if there is a reference to it at the old node. An object does not need an Address simply because it moves. For example, an object created and returned at the end of a remote invocation will not leave any trailing references as it moves across the network. However, an object which was refered to at its previous node will need an Address when it moves because, as in the second state, a reference now crosses a process boundary. When an object migrates it is replaced locally by its Address and the Forwarding object pointing to its new location.

2.1.2 Processes

A Process is an object that contains, at a minimum, a default Memory and an evaluation engine. It may additionally contain several objects, as well as pointers to other Processes and Memories. There is one Process object per Dreme process,

Address Information	
`Address String`	address string of object
`Parent Pointer`	pointer to Process from which Address was received
`Child List`	list of Processes to which this Process has passed copies of this Address
Forwarding Information	
`Current Memory`	last know location of the addressed object
`generation`	generation id for this forwarding object
Marshalling Information	
`Lock ID`	Unique identifier of the marshalling operation. If negative, then this is a write lock

Figure 2.2: Address Data Structures

i.e., a unique Process object per operating system process. (From now on we will refer to the Process object with an upper-case "P" and the actual operating system process with a lower-case "p.") The Process mediates communication with the outside world as well as giving Dreme objects a focal point for enquiring about the local environment. The default Memory is simply the process address space. Each Process has a host name and a process name, which uniquely identify this Process on the network. The Process object also keeps a list of Neighbor objects, which point to all known Processes to whom references exist.

The Process is not just a convenient abstraction; it can also be used to provide policy and security to the objects residing in its domain. The initial entry point from an object in one Process to the objects in another is the remote Process object, which can be queried to locate pointers to objects.

The Process provides four important and overlapping services.

1. It acts as a name server for remote objects through the `create-name` and `find-name` commands. The former allows a (possibly local) object to add a service and the latter allows a (probably remote) object to find the advertised service. The default behavior for both of these commands is to simply return

false `#f`, which provides no information.

2. It evaluates statements on behalf of external objects, through the `remote-exec` command. Depending on the origin of the call, the Process can choose to perform the request or not. The default behavior is to evaluate any request.

3. It negotiates for access rights on behalf of its objects when they move to a remote Process. The main rights to be negotiated are for access to `remote-exec` and for the composition of the *local environment* (described below). The default behavior is to perform no negotiation.

4. The Process decides on the default *local environment* of any mobile code which will execute locally. The local environment provides access to top-level services, as will be explained in the section on remote evaluation. The default local environment is the top-level environment of the Process.

With the default behavior, each Process looks to the rest of the world like a first-class environment, and the global address space makes the network appear to the programmer as a large, shared-memory, multiprocessor with a very slow bus. At the other extreme, the behavior can be set to provide no remote communication at all, or just a set of services with no mobility at all.

The Process object is a regular Scheme closure in the top-level environment. As the default behavior will not scale to systems where access must be controlled, the nature of the services actually provided can be set dynamically by replacing it with a new one using the `set!` command. However, no communication can occur without a Process object supporting the appropriate functions.

Communication among Processes currently occurs in plain text. The security of the communication would be considerably enhanced through the use of encryption and digital signatures. These features can be added to the handshake protocol between Processes. We will discuss encryption from time to time, but the addition of that feature would not significantly alter the rest of the discussion; we will assume that we can identify the sender of any communication.

Following closely in the path of the Internet, a Dreme Process has a three-part name:

1. *hostName* gives the Internet address of the host machine. This is currently the string representation, as opposed to the actual 32 bit number.

2. *procName* is a string giving the process name. The process name should be unique for the given host and is used by other Dreme processes to locate the port associated with that process. We use a name, rather than simply adopt the process id (pid) assigned by the operating system because it is highly unlikely that the same logical process will be assigned the same pid twice by the operating system if it is restarted.

3. *portNum* is a number identifying the actual port being used by the process. This is optional if the process actually has a unique name.

Internet utilities, such as `ftp` and `telnet` generally use just the host name with an optional port number. These utilities, however, are usually assigned to particular ports which are the same across the entire Internet. Dreme does not currently enjoy that luxury.

2.1.3 Neighbors

Suppose $\mathcal{A}$ is a Process. In order to communicate with other Processes around the network, $\mathcal{A}$ must know where they are and how to communicate with them. These pointers are encapsulated in objects called Neighbors. Neighbors are created either through a call to `make-neighbor`, giving the neighboring Process' name, or through the receipt of a message from an unknown Process. At start-up, the Process opens up a socket and assigns a thread to listen for connections. When one arrives, the two Processes engage in a handshake protocol during which each either finds or creates a Neighbor object for the other. $\mathcal{A}$ starts off with a predefined set of Neighbors, such as name servers; as $\mathcal{A}$ further explores the network or makes itself visible to other Processes, other Neighbors will be created.

A Neighbor created through `make-neighbor` initially contains just the Address of the target Process, a reference to a default local environment, and whether or not `remote-exec` is available. The latter two correspond to the permissions permitted for that Neighbor, and the default values are respectively the top-level environment and true `#t`. Were encryption used in the protocol, then the appropriate information would be stored here as well.

Since a whole neighborhood of mutually acquainted Processes might be started up simultaneously for a particular application, no attempt is made at creation to actually connect with the Neighbor (similar to the way Scheme's `letrec` creates identifiers when entering a scope but not assigning them values to support mutual recursion). At some point, though, a message will need to pass from $\mathcal{A}$ to a Neighbor, and at this point $\mathcal{A}$ will attempt to establish communications with the neighboring Process. If successful, the two Neighbor objects will be filled in with information about the connection. If the connection ever fails then the Neighbor object will raise an exception.

If a local function is called from another Process, then the identity of that Neighbor is placed in the global variable `source-neighbor`, and the identity of the continuation of the call (which, as we have seen, may not be the same) in the variable `destination-neighbor`. This provides one mechanism by which a Dreme

Process can control access rights from outside sources. The local environment, discussed below, is another.

2.1.4 Memories

Memories are places for inserting and retrieving Dreme objects. Each Memory is attached to a specific Process and has a name unique among the Memories attached to that Process. All object migration in Dreme involves moving an object from one Memory to another, even when an object is sent to another Process. At any given time, each Dreme object is either in a particular Memory of a particular Process or in transit between Memories. Each Process starts out with a default Memory, but can create additional ones as required. Memory was created as a separate abstraction in the face of numerous elements in the environment that can segment the locations addressable by a Dreme object, in particular the process address space and local file system.

Straightforward uses of Memories are providing checkpointing, persistence, and temporary storage of large data-structures using indexed files. For checkpointing, the entire current state of the in-memory executing Dreme process can be written to a file to be reread when the Process is next started up (this is the part of the closure of the current continuation which resides locally). Objects can be made persistent by `copying` them (rather than `moving` them) to and from Memories. (If an object is moved from one Memory to another, then its data is no longer preserved in the first Memory. On the other hand, if the object is copied, then both copies continue to exist.) When moving an object, Dreme generally moves the mobile portion of the object graph rooted at that object residing in that Memory, although checkpointing moves everything. When objects not in the Dreme base set (i.e., new C++ classes) are checkpointed, particular care must be taken to ensure correctness. Certain objects may need to go to specific Memories because of their underlying format. Likewise, care must be taken when using a Memory for temporary storage – if the granularity of the objects being stored is not well chosen, very little RAM may be saved.

In a more complex environment, however, Memories can be used to finely control the accesses allowed by foreign objects executing locally. Because the current Dreme implementation is an interpreter, a Memory is just a chunk of the process memory or local file system allocated by a function call. If we consider a Dreme system as being closer to an operating system, then we can consider representing segments of the virtual address space as Memories. A function's access rights to various Memories represents a particular memory map. Pages which are not mapped represent Memories which are completely invisible to that function. Sensitive elements of the local environment could be mapped into segments with read/write permissions turned off, meaning that all attempts to access those functions are trapped by the kernel. Only those pages with write permission could be used for creating new objects, so the function can only use a limited space. Finally, even having a pointer to an object of another user does not necessarily provide access, as permission to read the particular page is necessary. These aspects of the Memory abstraction have not been developed in the current implementation, but would fiture prominently in any commercial implementation of these concepts.

It has been argued [26] that interpreted code is somehow inherently safer than object code in the context of mbility, as the interepreted code is easier to control. I would argue that this conceit is false; the ability of malicious code, such as a virus, to harm a system depends on the access it is given to system resources, not whether it is interpreted or compiled. Control of the virtual memory mapping is sufficient to control access to any system resources; viruses are a problem for microcomputers using single-user operating systems because they give all executables full access to the entire system. The current version of Dreme is an interpreter not because that provides greater security, but because interpreted code is more easily ported to different architectures.

A separate class of Memory objects can help to provide a clean abstraction for the various external systems, such as flat files, relational and object-oriented databases (OODBMS), or even non-Dreme applications, with which a Dreme Process might need to exchange objects, so long as the interaction can be described in a top-down manner (Dreme requests an object of a Memory and receives one back) or as two co-routines (so to Dreme it appears as a function call). To retrieve an object, Dreme passes an Address, or some other construct, to the given Memory which identifies the desired object. The Memory then searches for the object and returns it to Dreme. For example, if the Memory encloses a relational database, then Dreme might pass in the text of an SQL statement. The Memory then passes back a `cons` cell whose `car` contains the first row of the result and whose `cdr` points to an Address, α, in the particular Memory. To retrieve the next row, Dreme passes α to the Memory, which uses this to identify the appropriate SQL cursor, get the next row, and pass back another `cons` cell like the first. The last call to the memory after the final row returns `'()`. Communication with an OODBMS might use that database's object IDs in addition to its query language.

2.2 Mobility

The last section sketched out the environment in which Dreme objects live. They come into existence in a Memory attached to some Process residing in a Node on the network. Over the course of an object's life it may be subject to a number of events. Most of these will be standard, as would occur in any other system, however sometimes a Dreme object will move from its current Memory to some other Memory in the network. This section describes how mobility is managed in the Dreme system.

Dreme objects are divided between mutable/immutable and mobile/immobile axes. Determining if an object is mutable or immutable can be determined statically. Immutable objects are freely copied around the network, as the copies will remain identical. Mutable objects, currently, exist in only one location at a time. References to them may exist elsewhere on the network, but these references must eventually point to the single copy. Whether an object is mobile or immobile (i.e., can migrate in a message or must stay in a single process) is decided by the run-time system with hints from the program itself. (An implementation is free to provide multiple copies of mutable objects if it can maintain the appropriate semantics.) Objects move either implicitly through access patterns (eg., as the return value of a function call), or explicitly as the parameter of a `move` or `remote-exec` command. Alternatively, an object may be `pin`ned, and therefore not move at all. (For immutable objects, replace `move` by `copy`.)

Once the decision has been made that an object will move, this must be done in an efficient manner. Moving a list one `cons` cell at a time will create excessive network traffic if the recipient will traverse the entire list, and not just some particular piece. Since most of the delay in a distributed computation is in network transmission time plus overhead, it is better to err on the side of over transmission. Therefore we need some straightforward default way to determine a reasonable size for transmission. As objects are either mobile or immobile, each object can be seen as the root of a graph whose interior nodes are potentially mobile and whose leaves either are pinned or are immutable and contain no further references. When an object moves, this whole graph moves (with immobile leaves being replaced by the addresses of the corresponding objects and the immutable ones by copies). The policies for designating an object as *mobile/immobile, mutable/immutable* will vary from system to system.

2.2.1 Mobility and Addresses

When to give an Address

We have shown it is unnecessary to give Addresses to all objects; only objects visible from outside the Process actually need addresses. The following conditions attempt to be parsimonious in the assignment of Addresses:

1. *Immutable* objects can be copied from Memory to Memory when not `pin`ned. Nevertheless they must be given Addresses if not all uses can be identified or if they may be tested for `eq?`-ness (physical identity). These conditions can be determined by lexical analysis. Two immutable copies of the same object on different nodes should be considered the same object and merged into one if they ever arrive on the same node; `eq?` should respond `#t` if two Addresses are the same. Numbers, characters, and booleans, which never require pointer comparisons, will not be given Addresses when copied. Of course, an implementation (or application) is free to `pin` an immutable object if it is deemed too large or too rarely accessed to copy.

2. An object which is the argument of a `move` or `remote-exec` command gets an Address (since clearly something refers to it).

3. If objects α and β reside on the same node, with β refering to α, then if β moves, α must have an Address before β actually moves, so that β can bring the Address to its new location and use it to refer to α. Note that β does

not necessarily receive an Address (for example, if it is the return value of a function call, there may be no local reference to it when it moves).

4. If a mutable object is referred to in the construction of a new object (such as with `cons` or `append`), or through some kind of `set!` operation, then it is marked as *referenced* (using a flag). If a referenced object is moved (directly or indirectly), then it receives an Address.

5. If all references to an object being moved are contained in the objects being moved with it, then it does not need an Address. An obvious example of this is a list where all external references are to the head. Another, more interesting example involves moving closures.

The first four conditions hand out Addresses conservatively. The fourth last, in particular, hands them out without consideration for the location of the referencing objects. The fifth condition allows the possibility of countermanding the previous one when the transfer is shown to be safe.

When an Object is Moved

When an object, α, is moved, it is replaced locally by the Address, augmented by a pointer to a *Forwarding object*. This latter object contains α's new location (i.e., Process and Memory ids) and a generation id (initially 0). Each time the object moves, the generation id is incremented at the new location; when comparing two Forwarding objects refering to the same object, the later generation is more up-to-date.

Object references propagate through the network either when an object moves or when a reference to it passes from one Process to another (in which case the Address is copied across the network). To support garbage collection, we will maintain all the copies of an Address in a tree routed at the object itself. (Although these copies cannot form cycles, other objects in the network can.)

When the Address pointing to object α is copied from node $\mathcal{A}$ to node $\mathcal{B}$, the Address at $\mathcal{A}$ keeps a pointer to $\mathcal{B}$ on its child pointer list. If $\mathcal{B}$ doesn't know of α, then $\mathcal{B}$ makes a new copy of the Address with the parent pointer pointing to $\mathcal{A}$. Otherwise, if $\mathcal{B}$ already has an Address for α (say from $\mathcal{D}$), then it compares generations. If the Forwarding object from $\mathcal{A}$ is from a later generation, then $\mathcal{B}$ switches allegiance (i.e., the parent pointer switches to $\mathcal{A}$). An additional message passes from $\mathcal{B}$ to $\mathcal{D}$, informing it of the change, so that the extra pointer can be eliminated, along with the new Forwarding object (the Address at $\mathcal{D}$ can use this to change its Forwarding object, but it must keep the same parent pointer, as there is no matching new child pointer anywhere). As a result, a Process has only one copy of an Address for a given object. On the other hand, when α moves from $\mathcal{A}$ to $\mathcal{B}$, the Address at $\mathcal{A}$ changes its parent pointer to point to $\mathcal{B}$. At $\mathcal{B}$, α places $\mathcal{A}$ on its child list and, if there is already an Address at $\mathcal{B}$, removes its parent pointer.

If the protocol is obeyed, neither parent pointers nor child pointers for a single object can result in cycles. In the case of parent pointers, a cycle can only occur if some Address, α has a descendant as parent somewhere. Suppose this is the case. Then that Address has switched allegiance to some pointer which is a descendant of itself. However a node will only switch parents if it receives a Forwarding object from an Address of a later generation than its own. If that is the case, then somewhere along the alleged cycle, some Address, β, switched allegiance to an Address of a later generation then α, so β is no longer a descendant of α, and there is no cycle. This holds even for the latest generation. The root of all reference chains for the latest generation is the object itself; its Address has no parent, and therefore cannot be part of a cycle. The (valid) child pointers are just the inverse graph, and therefore also do not contain cycles. This is important since a cycle would indicate some portion of the reference graph is cut off from the rest. If this were to happen through the behavior of some Process not obeying the protocol, then a certain number of Addresses would be cut off from the object. Other Addresses would still function correctly. Although this can be eliminated in a closed environment, it shows an unavoidable danger of using reference chains in an open network; there is an implicit contract among all the members of the chain.

This scheme is very similar to the Scion-Stub Pointer Chains proposed by Shapiro[54], but provides additional support for mobility, without which there is no need to be concerned for cycles. Emerald[50, 41, 28] uses addresses, but resorts to broadcast if an object isn't at the last know location. In the DEC Hermes[7] project, each object has a home site which know the current physical location, although this requires extra messages to keep consistent. Network Objects[5] maintain the child pointers, but not reference chains; each object directly knows everyone who refers to it.

2.2.2 What to Move

In contrast to objects in most systems, which are totally immobile, Dreme objects are mobile by default. When an object's value is requested by another Process, the object moves to that Process (possibly carrying some extra baggage with it) unless the object has been `pin`ned. Therefore it is more interesting to enumerate the objects which are not mobile.

1. Numbers, booleans, characters, symbols, and the Dreme default function set, are all immutable. They cannot be removed from a Dreme process, as this would create severe implementation and semantic problems, but they can be "copied". References to them in one process can easily be replaced by

references to the local versions in another machine. Messages contain sufficient information to locate or create the appropriate objects on the receiving machine.

2. C++ functions are currently both immutable and immobile. With a run-time linker they may eventually be copyable, if the underlying O/S and hardware are compatible. C++ objects, on the other hand are mobile if the appropriate methods have been overloaded. If the receiver does not know how to handle the incoming object, an exception is thrown to the sender.

3. Monitors are immobile by default. The justification here is that monitors are used for concurrency control among several threads in the network and should normally reside at the point where this is needed, rather than spending time in transit. This is an implementation choice that could be rescinded if found to be overly restrictive. In any case, it is possible to `unpin` a monitor to move it.

4. Continuations are also immobile by default, although they can also be moved when necessary. Continuations usually represent a point in a particular computation in a particular thread. The point of having a pointer to a remote continuation is to be able to return a value to that computation at its location, even if this seems to break the symmetry of the system. It is desirable that a continuation, some of its environment, and a copy of the function, all reside in the same location, or else the Process will need to go over the network to locate each identifier and fetch each successive instruction.

5. Objects explicitly `pinned` by the program are immobile. In the case of objects created by the program, only references to them can be exported in messages,

but in the cases of the immutable classes mentioned above, there is no change in behavior. Likewise, objects `unpinned` by the program become mobile if they are created by the program. C++ functions still do not move under the current implementation.

In the case of an invocation, the expression in the operator position of the S-expression is considered to evaluate to a function pointer, as in C/C++. If the function is not local, then this is its Address. If the operator is an Address, then the whole invocation is packaged and shipped to the site of the function, rather than moving the function to the invoker, even if it is mobile. This allows closures to act as communicating objects. When this is the case, parameters that are remote references are also shipped as is; otherwise the objects would need to be shipped twice, once to the invoker and once along with the message to the function.

Moving Functions

When moving a function from one Memory to another, we need to know not just how to move the function, but also how much of its closure to move as well. Let α be an identifier whose value may change over the course of the computation. Since α is mutable, it must be bound to a particular location, β, in the network. If a function within the scope of α moves, should β move as well? Certainly, if the only references to α are within that function, then β should move as well.

The code fragment in figure 2.3 gives an example of this. Each time `server` is called, a new client function is created and returned. The clients share access to the identifiers `a, b, c` and `c-1`, but each has a private set of `aa, bb, cc`, and `dd`. The latter set should move with the client, whereas the first set should remain at the server location (or be replicated with an appropriate consistency protocol).

The correct behavior can be derived by assigning to each identifier and each

```
1 (define server
    (let ((a (lambda (...) ...))
          (b ...)
          (c ...)
5         (c-l '()))
      (lambda (...)
        (let ((client
               (let ((aa expr1)
                     (bb b))
10               (letrec
                     ((cc expr2)
                      (dd (pin expr3)))
                   (lambda (e f g)
                     ... code referring only to a, aa-dd, e-f ...)))))
15        (set! c-l (cons client c-l))
          client))))
```

Figure 2.3: Deciding what to Migrate

function a nesting level. The number of scopes which a function brings with it when moving is one less than the difference between the level of the function and the level of the identifier to which it will be assigned (if the outermost scope is at the top-level, then it is assigned to level zero and the outermost scope is at level one). In the example, `server` is at level 0, the `let` in line 1 is at level 1, the `lambda` in line 6 is at level 2 (as are `a`, `b`, `c`, and `c-l`). The identifier `client` is at level 4 and the `lambda` in line 13, which is returned by the function, is at level 7. Since $7 - (4 + 1) = 2$, the returned function brings two outer scopes with it (`aa`, `bb`, `cc`, and `dd`) as desired.

This analysis, as written, only holds for identifiers. From the code fragment, we can see that each remote client, in turn, will bring an example of `b` with it, but the identifier `dd` will point to an Address – the actual object is kept at the location of the server (although with no local references to it). An extension of the same analysis can be used to determine if the results of `expr1` and `expr2` need to be given Addresses when they move. This requires determining if there is a reference to the returned value at an outer scope.

To the extent that the local environment is not accessed, this is really an implementation issue, as global addressing will maintain the right semantics wherever the identifier is located. Nevertheless, there are definite performance considerations associated with the placement of objects around the network.

2.2.3 How to Move in a Concurrent Environment

As mentioned, when an object moves, it brings along the mobile graph of which it is the root. Turning the graph into a message requires a marshalling process to linearize the message and place it in a buffer. Complexity arises from two sources:

1. Other threads in the Process may attempt to access objects while they are being marshalled, including other marshaling processes.

2. The message graph may not be a tree, so certain objects will appear more than once in the traversal.

The dangers from concurrent marshalling operations are inconsistent behavior (an object might end up in two Memories when it should only be in one, or alternatively, an access to an object might precede the object's arrival) and deadlock. The key to avoiding these problems is ensuring one marshalling operation is always able to complete and then ordering message transmissions by their dependencies. Concurrent threads of control present a danger in that data which is being marshalled could be destructively altered, so that the data transmitted is inconsistent with the current state of the system. This is similar to the serialization constraint placed on database transaction systems. The simplest solution is *stop-and-copy*: when a marshalling operation commences, halt all other processing until it completes. This is not an adequate solution, as there is no fixed limit to the size of a message, and two processes trying to transmit to each other would immediately deadlock.

The rest of this section describes an algorithm which resolves these issues while still allowing some concurrency during the marshalling and demarshalling operations. For the sake of exposition, it is described as requiring three traversals of the objects graph, although it is possible to combine the first two traversals into one by increasing the communication among marshalling operations. We will also assume that all interior nodes have Addresses; any that don't can be assigned temporary ones for the duration of the operation. This is a "worst case" algorithm; better performance is possible when the visibility of objects is known.

The First Traversal

The first traversal determines the contents of the message and resolves the concurrency issues among the operations. Contention among marshaling operations and other threads over access to objects is handled by having each operation first place a read lock and then later a write lock (in the parlance of database transactions) on the objects it will move. This initially prevents other threads from changing the state of objects which will be included in a message and later prevents any access until the object arrives at its destination. Each marshalling operation is assigned an index and contended locks are assigned to to the lower indexed operation. This way, the only dependencies among messages will be from higher indexed messages to lower indexed ones, meaning that the lowest indexed message will always be free to be sent. In addition, each marshalling operation keeps a list of the operations it depends on, so each message knows the earliest point at which it can be transmitted. The traversal proceeds as follows:

1. At startup, each marshaling operation is assigned an index.

2. The operation traverses the object graph. If a node in the graph is an immutable base type (number, character, or boolean), then it is a leaf of the traversal, and can be left as such. Otherwise the operation attempts to read lock the object. The lock operation is as follows:

   (a) If the item is unlocked, then set the `lockId` field of the Address to the index of the operation. Return Success.

   (b) If the item is locked by an operation of lower index, then put that operation on the current operation's dependency list. Return Fail.

   (c) If the item is locked by an operation of higher index, then place the current index in the marshalId field and place the current operation in the other operation's dependency list. Return Success.

   (d) If the item is locked by the current operation do nothing. Return Fail.

   If the lock procedure returns Success, then traverse the newly-locked item. If the item is mobile, then update the Forwarding object to point to the target of the message.

3. After traversing the graph, the current operation waits for all marshalling operations of lower index to complete their first traversal (until then, the set of objects locked by the current operation is not stable), and then complete it as well.

Once an item has been locked by a marshalling operation, it is *read only*. Any destructive operation on a locked item causes the offending continuation to be blocked awaiting completion of the message transmission. This way the contents of the message will always be correct with regards to the current state of the system, but other operations are not blocked. The obvious implication is that an applicaiton must be very careful about moving objects involved in real-time operations. At the same time, the contents of the message is now stable.

The Second Traversal

The second traversal determines the size of the message, how many objects, if any, are at the head of cross edges or back edges in the graph, and how many are actually locked. As the nodes are traversed, they are marked and their sizes summed. Nodes that are encountered a second time are given increasing indices from zero in the marshalId field. At the end of this traversal, the operation knows how large the buffer needs to be, how many repeated items it will contain, and how many items it has locked.

The Third Traversal

Before commencing the third traversal, the operation creates the actual message object. This structure contains the fields described in figure X, but the most important for our current purpose are the `indexSize`, which will contain the number of repeated objects, and `msgBuffer`, which will contain all the objects moving to the destination. Another array the size of the number of locked objects will be used to unlock them at the end.

During the third traversal, the objects are placed in `msgBuffer`. Each object type has its own particular representation. Immobile objects or objects locked by other marshalling operations are represented by their Addresses (the Forwarding information will be correct regardless of the state of the other operation because it was changed in the first traversal). When an object is encountered for the second or later time, its index value is used instead. Each time an object locked by the current operation which is to move is encountered, the lock is raised to a write lock, meaning that any other thread attempting to access that object will be blocked awaiting successful completion of the message transmission.

After the third traversal, the message is ready to send. Each message must wait until all messages it depends on (as determined during the first traversal) has been sent. Once a message is successfully received and the objects reconstituted, each moved object is replaced by its Address, all the locks are relinquished and all the threads blocked by the message operation can be restarted.

Demarshalling

Demarshalling involves a single pass through the buffer. Pointers to objects involved in cycles are maintained in an array. Since the buffer contains a depth-first traversal, the first time such an object is encountered, it will contain the body of the object. Subsequent references will only contain the object's index. As each object is entered by the demarshalling operation it is write locked. The lock is lowered to read when the object is exited. Finally, when all objects are locally reconstituted, all locks are lifted and all threads restarted.

In the underlying implementation, marshalling and demarshalling are accomplished by recursive descent using C++ virtual functions. This allows C++ classes linked into the implementation to benefit from mobility as well (although the object must be implemented on both sides).

The algorithm is presented for the general case, on the assumption of concurrent traversals with little information about the objects encountered. In a more sophisticated system, information can be derived about the accessibility of objects to limit the amount of locking. For example, if there is only one access path to a

`Remote Execution`	Message contains a thunk to be evaluated by the receiving process, which will send the result to the continuation of the call
`Asynchronous Execution`	Message contains a thunk to be evaluated by the receiving process. No response is expected
`Remote Invocation`	Message contains a function pointer and one or more parameters. The function is to be applied to the parameters, and the result sent to the continuation
`Asynchronous Invocation`	As above, except no reply is expected.
`Return Value`	Message is the reply from a `Remote Execution` or Remote Invocation message

Figure 2.4: Message Types

`Sender Address`	address string of sending object
`Continuation Address`	Address of the continuation of the message (including forwarding information)
`Sequence Number`	all messages between two Processes are numbered. Only garbage collection messages are necessarily ordered.
`Message Type`	The message is one of five types
`Index Size`	Number of items encountered more than once while marshaling the operation
`Index`	locations in buffer of all the indexed items
`Message Buffer`	Contents of the actual message

Figure 2.5: Message Buffer Elements

subgraph, only the pivotal object actually needs to be locked, or if the object was created by the sender, no locking may need to take place.

2.3 Remote Invocation

So far we have discussed how objects move about, but not how they are used. Eventually some node $\mathcal{A}$ will want to access a value residing at a remote node through local object α. When this occurs, the remote location sends the message to the location indicated by the Forwarding object. After some time, a response will arrive from Ω, the node containing the target of the invocation. The structure of a message and of the response is shown in fig ?. From the perspective of Addresses and Forward references, we can break this into four cases:

1. α's parent pointer and Forwarding object both point to Ω. In this case, the Address at Ω also has a pointer back to $\mathcal{A}$ and can tell (from the message) that the message took no indirect hops. Ω need return only the answer to the continuation of the call.

2. Neither α's parent pointer nor Forwarding object points to Ω. In this case, at the first hop, the message is marked as *forwarded.* When it arrives at Ω, the Address at Ω will create a child pointer to $\mathcal{A}$. If the continuation of the call is at $\mathcal{A}$, then Ω will pass a copy of the Forwarding object directly back to $\mathcal{A}$ with the response. Since this structure is of a more recent generation than the one currently at $\mathcal{A}$, α will change both its parent pointer and Forwarding object. If the continuation of the call is not at $\mathcal{A}$, but at another node, $\mathcal{C}$, then Ω will send the response to $\mathcal{C}$, rather than back along the call chain to $\mathcal{A}$. In either case, Ω will send the new Forwarding object back along the call chain, where each node will change its Forwarding object but not its parent pointer (since the Address at Ω has not created pointers to any of them). This second step is the key improvement of this algorithm, as explained below.

3. Only α's Forwarding object points to Ω. If the Address at Ω doesn't have a pointer to $\mathcal{A}$, then it adds one to its list. Ω sends an update parent message to $\mathcal{A}$ (which may piggy-back on the answer), and α updates its parent pointer. The answer, again, goes to the continuation, which might be in $\mathcal{A}$

4. Only α's parent pointer points to Ω. This means that the object has moved to Ω since the Address arrived at $\mathcal{A}$. The invocation will proceed as in step

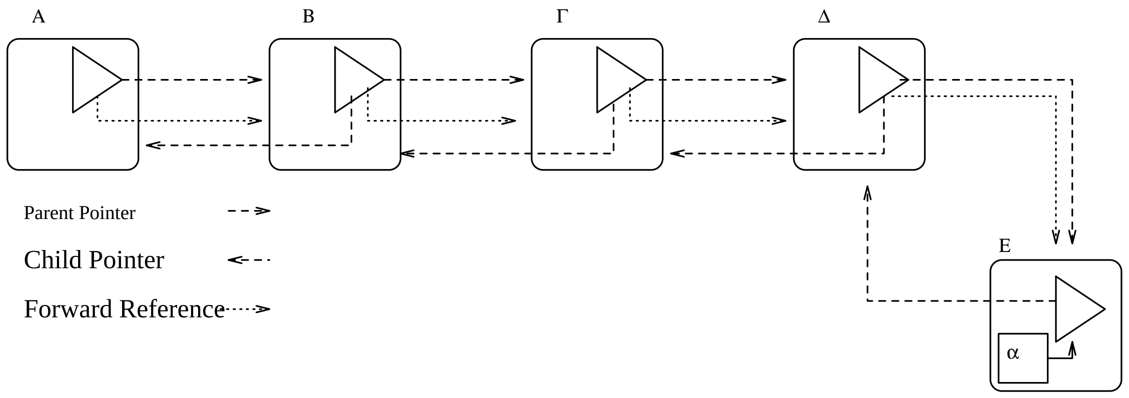


Figure 2.6: Object α migrates from A to E.

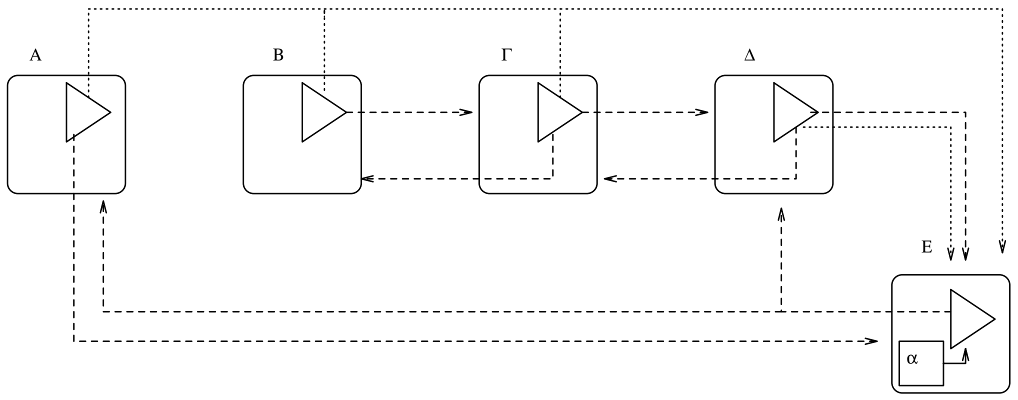


Figure 2.7: After α is invoked from node A.

2.

Figure 2.6 shows the case where object α migrates from node A through to node E (or a reference to α migrates from node E to node A). Figure 2.7 shows the situation after a message passes from node A to the object.

The location mechanism here is complicated by the inability to resort to broadcast in a very large network, forcing the algorithm to track its target one step at a time. The use of path compression, as in [58], for longer chains means the amortized worst case number of messages is $n\log n$ if the reference graph is a tree, where n is the number of links added by Address propagation plus object movement. The *invocation*, however, only traversed $n+1$ links: from the source to the destination, and then to the continuation. Where newer generations of Forwarding objects encounter older ones, the path is compressed automatically.

The reference mechanism in SSP is similar, except that the forward references of intermediate Addresses (called *weak locators*) are not updated, which can lead to a large number ($O(n^2)$) of messages in the case of long chains with bad access patterns. LII also uses path compression, but does so by treating the sequence of references (called *tads*) as a stack, so the messaging system cannot take advantage of distributed tail-call optimization, which eliminates half the round trip. Path compression is also important because it allows interior nodes to become leaf nodes, which makes them easier to garbage collect.

2.4 Remote Evaluation

In keeping with Scheme, Dreme identifiers are generally lexically scoped. If this were completely true, however, then objects executing on a remote Process would have great difficulty accessing local resources. All such resources would need to have been negotiated between the object's originating Process and the current host Process, and then placed in the scope of the mobile object. Unfortunately even this expedient fails when the object moves again; there needs to be a solution which provides a mobile object with access to local resources regardless of the Process it is currently executing on, while allowing the local system to control just what that access is. There are two main reasons for requiring this:

1. Code originating remotely may be buggy or malicious. It is important that other objects residing at the same host be protected from harm. Of course, even locally developed code might do damage, particularly if it is executing in a persistent environment which must provide services continuously. Any

mechanism for controlling code of remote origin can control code of local origin.

2. Lexical scoping means that any information accessed by the local function is potentially visible to other objects in the originating Process. Because it is not generally possible, and (given the overhead it would require) not generally desirable, to constantly monitor the behavior of every local executing function, preventing sensitive information from being accessed in the first place is probably the best way to keep it private.

To overcome this problem, Dreme introduces a limited form of dynamic scoping called a `host escape`. When a function arrives, it is assigned a *local environment*. The contents of this environment is decided by the Process object, which may simply be the top level environment (providing no control over the function's execution), or might be the result of negotiations between the sending and receiving Processes. During the course of execution of the function, any identifier with a prepended percent sign (`%`) is evaluated in this local environment rather than in the lexical environment. In this way, for example, a call to `write-line` writes a line at the originating Process, while `%write-line` writes one on the current host Process (if `write-line` is, in fact, in the function's local environment). Assigning the local environment is the link phase in receiving a remote object.

Members of the top level environment present a general difficulty for Lisp-type languages because top level variables are all mutable and it is impossible to determine lexically all the sites at which they may be changed, since the user is capable of changing them at any time. Nominally they must always be checked before being used. This represents a problem for compilation, as it means that the pointer to the function must always be checked, and it is a problem for a distributed Dreme program, as it means that each reference to a Dreme default function requires traveling over the network, even though the exact same function is available locally. We handle this by dividing the default function set into a *primitive* set, such as `car`, `cdr`, and `cons`, and others, such as `file-open`, or `display` which can adversely affect the local environment (we assume that the local process can control the local time and space usage of objects). Primitive functions are treated like numbers and booleans; they give the appearance of migrating, but they really exist the same in every Process. Other default functions are `pinned`; they cannot be moved, and so can only be called on the site from which the object originated. Local access to these latter functions is only available through the host escape mechanism.

For each Process, the local top-level environment is assigned as the default local environment for that Neighbor (i.e., itself) and for the default set of Dreme functions. When a message is received from a Neighbor, as the message is reconstituted locally, any functions encountered are given the default local environment assigned to that Neighbor. Whenever a new function is created, it is assigned the current local environment. Since all functions entered at the command line or read in with `load` are created from the read-eval-print loop, or some function created from it, all functions of local origin will have the top-level environment as the local environment unless explicitly specified otherwise. Likewise, any locally resident code of foreign origin is only accessible through functions brought from the foreign Process; any new closures will inherit the appropriate local environment. The local environment of any function can be changed with `change-local-environment`, but this is not necessarily accessible to foreign code. As mentioned before, the same mechanism can be used to give untested code access to a test environment, instead of the actual one, during development and testing. An attempt to access a non-existent identifier on this environment first generates an exception to a local exception handler, which can then turn the exception over to the normal handler for the executing code.

This simple scheme provides the local system with extensive control over access to the local environment by locally executing code. The decision to assign by Neighbor assumes that the inclusion/exclusion decision for code from a particular Neighbor generally holds for all such code, since any exposed information can be sent to that Neighbor by any piece of code with that origin. The contents of the local environment can be negotiated between Processes, and the local Process always has access to this environment, and therefore has the ability to alter or expand it. Nevertheless, this mechanism, by itself, is not sufficient to deter a determined opponent and needs to be integrated with a reliable means of authentication, such as one based on digital signatures.

2.5 Concurrency constructs in Dreme

As soon as more than one (physical) processor is available to a system, the possibility of concurrency arises; Dreme is scarcely the first Lisp/Scheme derivative to confront this issue. Concurrency breaks down into two aspects, creating it and controlling it. We will examine the Dreme approach in the reverse order.

2.5.1 Controlling Concurrency

The main Dreme construct for controlling concurrency is the `monitor`. A `monitor` is a lambda expression which serializes entry; once a continuation occupies the `monitor` any other continuation that attempts to enter is blocked until the currently executing continuation explicitly calls `release` to allow one of the waiting continuations to enter the `monitor`. `Release` must be called explicitly, as tail recursion

```
(define release-proc #f)

(define monitor-proc
  (let ((mon (let ((flag #f))
               (monitor ()
                 (if (not flag)
                     (set! release-proc
                                 (lambda () (release)))
                     (.... monitor-body ....))))))
    (mon)
    mon))
```

Figure 2.8: Monitor with an escaping `release`

elimination makes it difficult to determine exactly when a scope has been exited, although it would be possible to ensure this with a wind/unwind construct[14].

Dreme `monitor`s are lexically-scoped but are weakened by the presence of first-class functions. The thread inside the `monitor` can expose the associated `release` statement outside its scope. This is demonstrated by the code fragment in figure 2.8, which creates a `monitor` and then assigns access to the `release` call to the globally available `release-proc`.

The underlying implementation contains a few critical sections, however these are hidden from the application programmer. It is hoped that applications will not need to create critical sections, as these will unilaterally stop all threads from all applications. `Monitors` can be used to create critical sections among the threads of a single application.

The other means for controlling concurrency is through a call to `rep-loop`, the Dreme read-eval-print loop. This essentially terminates the calling continuation, as it never returns, but either handles the next communication from the environment (such as a command line entry, a user interface event, a message from another

Process) or schedules a waiting thread.

2.5.2 Creating Concurrency

Concurrency in Dreme is created by one of the following methods:

1. Assign a continuation to a port. Concurrency requires nonblocking communication with the external environment. Nevertheless, many applications wish to interact synchronously with the environment. Dreme assigns a continuation to the port and calls it when input (or output) is possible. A thread can choose to block at the call, or continue to execute. In the latter case, a new thread is effectively created to handle the communication. Communication with Neighbors is handled in the same fashion.

2. Assign a continuation to a User Interface event. This is analogous to the previous case, except that (at least in the case of X Windows) all events are coming over the same port.

3. Create a local task to be evaluated asynchronously by calling the scheduler (called `tasker`). Tasks may be added to the front of the list by calling `(tasker 'front` *thunk*`)`, or to the back. *Thunk* is a parameterless function encapsulating the task to be performed. This function returns immediately, and *thunk* will be scheduled to be evaluated.

4. Invoke a function asynchronously. If the function is local, then this behaves essentially as in the previous case, otherwise the invocation is sent to the destination and the local thread continues to execute.

5. If the Process has a preemptive scheduler, then threads will be interrupted and rescheduled when their quanta expire. Although this is not strictly required

by Dreme, it is necessary if the Process is going to give any guarantees of service to locally executing code. The current implementation uses a variation on engines [17] to implement preemptive scheduling.

All of these add to the number of threads being controlled by Dreme Processes, although the additional threads may be remote. Although preemption is not fundamental to Dreme, the scheduler is, as it is otherwise impossible to implement certain concurrency constructs.

The automatic parallelization of Scheme programs is a well-researched topic beyond the scope of the current work[23]. Any work done in that area is immediately applicable to Dreme, particularly if the parallel program is to be run over a network of workstations, as opposed to a single, shared-memory machine. Although the development of Dreme emphasizes distributed applications, instead of concurrent ones *per se* (a distributed application can be just as serial as a non-distributed one), an automatic parallelizer could take an ordinary Scheme program, parallelize it by adding Dreme commands for distributing the application, and then run it over a network. By drawing a strong separation between the location dependent and location independent elements of an application, Dreme routines to handle load balancing, error recovery, etc., could be built underneath the original application to manipulate the parallel pieces.

2.6 Distributed Programming Languages and Systems

Distributed systems is a large and constantly growing area of computer science. In this section we will concentrate on distributed languages, and particularly on languages similar to Dreme, such as such as Emerald, Obliq, and Telescript, which explicitly support mobility, as well as actor languages and distributed lisp implementations. We will also examine Sun Microsystem's Java, which promises to have significant impact, despite its novelty.

When development first started on Dreme, the only significant languages explicitly supporting physical mobility of objects in a network were Emerald [50, 41, 28] and, to a lesser extent, Eden [2, 6], which we will examine first. A number of systems have come into existence contemporaneously with Dreme. Excluding Telescript [26], about which so little is known, the most sophisticated, and the most similar to Dreme is Obliq [10, 9], an object-oriented language which also uses lexical scoping to support correct IPC.

2.6.1 Eden

Eden [6, 2] is an early example of an object-based distributed system. Although originally intended to be implemented as an operating system, it was eventually developed on top of Unix. The Eden programming language (EPL) was developed as a superset of Concurrent Euclid, itself an expansion on Pascal to add support for processes, modules, and monitors (in fact, an earlier implementation of Eden was programmed using Pascal; movement to a concurrent language superset became evident through the inability to express parallelism and invocation in Pascal).

The essential characteristics of Eden objects are that they:

1. are *large-grained* objects occupying an entire Unix process. In addition, each object must contain all the Eden support code, which might be an order of magnitude larger than the code for actually implementing the object.

2. are accessed by *synchronous* invocation. Invocations are type-checked partly at compile time and partly at run time.

3. are referenced by *capabilities*, which are composed of the unique identifier of the object and a set of sixteen rights. The unique identifier is used by the system to locate the object when an invocation is made.

4. conform to an *Edentype*, which specifies the interface to the object. An abstract Edentype might be implemented by any number of concrete Edentypes.

5. are *mobile* and can move from machine to machine without their clients being informed. It is possible to refer to specific machines; they are called *nodes* in Eden and can be accessed by capability, as can all other system objects. Mobility, however, is provided only to quiescent objects not currently responding to invocations.

6. are *active* at all times. Since Eden objects are written in EPL, they will have one or more active processes, although they might temporarily deactivate themselves to conserve resources. Synchronization inside objects is provided by monitors.

7. completely encapsulate local data.

8. potentially *persistent* through checkpointing.

Although EPL is statically type-checked, capabilities, interestingly enough, are not. When a capability is declared, the user specifies the abstract Edentype that it is for. The compiler will check that all uses of the capability correspond to the alleged type. At run time, however, the capability will be assigned to some Eden object, which may or may not correspond to the desired Edentype. The run-time system will only check to make sure that the invoked operations are supported by the referenced object. This unusual loophole in the type system is considered necessary for supporting system extensibility in the absence of inheritance. Otherwise a capability in a continuously running program could not be assigned to an object of an Edentype that was not created when the program started. In fact, a given Edentype might conform to any number of other abstract Edentypes, but no means is given to determine this conformity. This issue will be resolved in the typing system for Emerald, discussed below.

Eden has a sharp distinction between system objects, which are large, universally locatable, accessible using capabilities, etc., and any other object. This leads to Eden objects being used "sparingly". Other uses of concurrency are built from the Concurrent Euclid subset of EPL; a concurrent construct must be written twice if it is to be used both locally and globally. Again, this problem will be addressed by the Emerald system.

2.6.2 Emerald

Emerald [50, 35, 28] is an object-based distributed language (although the complexity to support the language brings it almost to the level of a distributed operating system) whose most innovative feature is its complete support of mobility for objects of every type and granularity. Associated with this is an explicit recognition of location. Most concurrent languages, in fact, completely finesse the question of where threads run. This makes sense for a language for a uniprocessor, like Concurrent C++, or for a shared memory multiprocessor, like PRESTO, where all locations look exactly the same. In such situations, concern with location is irrelevant at best, and positively damaging if it forces the programmer to make compile time decisions that can only be made optimally at run time. In a network of heterogeneous systems, however, location can be very important. Closely cooperating components should be placed near each other to minimize network delays;

components communicating with the outside world, such as device controllers, may have to reside on certain equipment. And in some situations it might be necessary to move them around. In addition, there are some interesting aspects to the Emerald language which bear mentioning as well.

Emerald has five primitives for implementing mobility:

1. *Locate* an object by finding the node it resides on.

2. *Move* an object to a given node or to the node that another given node is presently on.

3. *Fix* an object at a particular node so that it cannot subsequently be moved.

4. *Unfix* an object so that it can be moved.

5. *Refix* an fixed object at another node by atomically *Unfix*ing, *Moving*, and *Fixing* it.

In addition, Emerald supplies *call by move* and *call by visit* as parameter passing modes, along with *call by object reference.* In the first case, the parameter object is colocated with the invoked object and remains there after the invocation terminates. In the second, the parameter object is returned to the node it was at before the invocation. Since invocations are synchronous in Emerald (i.e., the executing thread physically moves to the remote node) and the caller will not be accessing the parameter while it is moved, this is quite useful, since the cost of piggy-backing some information on the invocation is much less than additional round trips while processing the invocation. In some cases, as in mail delivery, the object would have been the subject of a later *Move* command anyway.

The Emerald language does not have a class construct or inheritance. Instead, Emerald uses types and object constructors to convey the same, or similar, properties. In traditional Object Oriented languages, classes and inheritance provide two services: they provide common interfaces, and they support code reuse. Emerald takes the former capability and embeds it in its typing system, but rejects the latter. Code reuse assumes that similar interfaces imply similar construction, and so the same function can be used on any object of that class. (C++ is particularly egregious in this regard. Friend functions or friend classes allow uncontrolled access to the internals of objects of the class. Even overloaded binary operators do the same.) The Emerald compiler, however, must be free to modify the internal construction of various objects to ensure mobility. This is transparent to other objects, since all interaction is through the unchanged public interface. This has other benefits, since it allows developers to be as free as necessary with optimizing operations for different environments (or languages) or changing them with subsequent releases without worrying about making old code (which might still be running) obsolete.

An object, A, can be used wherever an object of type T is expected if the type of A, S, is conformant with T (S o> T). There are four conditions for such conformance:

1. S provides at least the operations of T (S may have more operations).

2. For each operation in T, the corresponding operation in S has the same number of arguments and results.

3. The types of the results of S's operations conform to the types of the results of T's operations.

4. The types of the arguments of T's operations conform to the types of the

arguments of S's operations.

The first three imply that anything that can be asked of a T can be asked indistinguishably to an S. The last says that if an S is acting like a T, no argument will be passed to it that cannot respond to messages that an S would send it, given the type of object that the S object expects.

Emerald types are first class objects and can be passed around as parameters and be declared as variables, with the proviso that all operations on them be resolvable at compile time. The type is a collection of *Signatures* (where *Signature* is an Emerald type) and is created by invoking a type constructor with a number of function declarations. The generality of this scheme means that not all type checking can be done statically at compile time; the run-time system will still need to check conformance in some cases. The error, however, will be caught at the invocation level, and not by some error in the middle of an operation. Emerald does not currently have an exception mechanism to handle these cases.

Emerald objects are created by calling an object constructor and assigning the value returned to some program element. This allows the creation of both one-of-a-kind objects and classes of objects (by wrapping the constructor in another function that calls it as requested and returns the new objects). An interesting example of this is the *Array* constructor, which is a routine which exports an *of* method which expects an abstract type parameter and returns an object that exports a *Create* method, which returns an object with various array methods. If *Create* also supports a *getSignature* method, then it is an abstract type and an integer array is declared:

```
var IntArr:Array.of[Integer].Create
```

Object location and mobility demand significant run-time support and impacts even the structure of objects. Each object is composed of four components:

1. A network unique *name* or *id*.

2. A *representation*, which is the data stored in the object. User defined objects contain references to other objects.

3. A set of *operations*, some of which are exported and can be invoked externally, some of which are private to the object.

4. An optional *process*. Objects with processes execute independently of each other and of any invocations on the associated objects (although certain operations may be placed within *monitors*, allowing for some concurrency control.

Note that, conceptually at least, each object carries around not only its data, but all the code for its operations. Each object is assumed to have a completely different implementation, so even objects from the same constructor cannot access each others internal data. This allows the compiler to actually create different implementations for the objects returned from the same constructor, depending on their potential uses. Emerald divides objects into three categories, according to how the compiler determines they might be used. These are:

1. *Global Objects*, which can move around the network and be called by external objects not known at compile time. They have global scope and so references to them can be passed around. They are allocated from the heap by the kernel, since they may need to move. An invocation of one might require a remote procedure call.

2. *Local Objects* are local to some other object. References to them never go beyond the local scope, so they are never called or moved except in conjunction with the outer object. They are allocated on the heap by compiled code.

3. *Direct Objects* are local objects allocated directly within the enclosing object, such as built-in types or simple objects.

Since they are all accessible only through their interfaces, the compiler can choose to implement an object in any of three ways. In terms of the physical representations, these are direct for *Direct Objects*, indirect for *Local Objects* and doubly indirect for *Global Objects*. In the latter cases, the local pointer has the address of an area of memory whose first word has a *tag*, a *G* bit and an *R* bit. If the *G* bit is set, then the object is global, and the area in memory is an *Object Descriptor*. The code then checks the *R* bit. If it is set, then the object is on the same node, and the next word of memory is a pointer to the *Object Data Area*, which becomes the target of the invocation, otherwise the system traps to the kernel for a remote invocation. If the *G* bit is not set, then the object is local and the memory area is the *Object Data Area*.

Mobility is an essential part of the Emerald system. Moving a simple data object is straightforward. The kernel uses a template to build a message containing first the data area and then mapping information to allow the receiving kernel to map location dependent addresses. Since the data area can contain references to other objects, information on them must also be sent. For global objects, the unique OID, forwarding address, and address of the object descriptor on the sending node ar included. For local objects, the appropriate information is sent recursively. The receiver allocates space for the new global object, builds a translation table for all the objects received, and then uses their templates to traverse the structures, correcting pointers. For unknown global objects, new descriptors are built using the information received.

The process becomes much more complicated when there are invocations active in the object, since they are moved as well. When an Emerald process is preempted, the list of outstanding activation records is scanned and they are linked to the objects in whose bodies they executed (at each return the process must unlink the activation record, although most of these should occur between preemptions, so no work should be done except checking a flag). When an object is moved, the list of activation records is scanned and they are packaged similarly to the data portions. If an invocation record is moved from the inside of a stack, then the stack is split into three parts, with one being sent, and all three being modified to appear as if remote invocations had occurred. Finally the list of invocations must be scanned to locate any spill registers that the moving portion needs that are stored elsewhere on the machine, and any spill registers that it contains are linked into the appropriate place.

2.6.3 Obliq

Obliq, developed by Luca Cardelli at DEC SRC, is an object-oriented programming language supporting mobile closures in a network. Obliq is an interpreted language written in Modula-3 that uses *network objects*, a distributed object system, also developed at DEC SRC, for Modula-3 objects.

Objects in Obliq are a collection of named fields, where each field is a *value*, a *method*, or an *alias*. Values and methods have their usual meaning. An alias is a reference to a method in another object, so that invoking the method in the ostensible target of a method invocation actually invokes the method which it aliases. The Oblique object system contains no classes; inheritance is approximated through the *cloning* an object, similar to *Self*, where objects can serve as prototypes for other objects. A single object can be created by cloning several other objects.

As a distributed system with some mobility, Obliq provides a network-wide distributed memory similar to Dreme. The global address space is divided into *sites*, which represent physical address spaces in the network. Each site contains a number of *locations* where mutable values are placed. Locations are fixed to particular sites and do not migrate. Transitively, entities that encapsulate locations also cannot migrate. These include:

1. *objects* which are both bound to locations and contain locations for their updateable fields.

2. *arrays* since the array elements are mutable.

3. *variables* as their values are also updatable.

In other words, mutable Obliq entities, unlike mutable Dreme entities, are fixed to their sites.

Unlike mutable objects, Obliq closures are mobile, but differently than are Dreme ones. An Obliq closure consists of its code and a table of its free variables. The code can be transmitted, but not the variables. In Dreme, objects are ordinarily transmitted if they are completely encapsulated in the closure, as they are no longer accessible on the originating node. There are two possible reasons for this difference:

1. Obliq depends on Modula-3 network objects, which do not migrate once created. Therefore, once a location is created, it is fixed. Since all object management in Dreme was developed with the interpreter, Dreme does not have this restraint.

2. Obliq considers objects to be structures with named fields. Although Dreme does not currently have an object system, our preferences are towards treating objects as a particular type of closure, where an object is, above all, the locations it encapsulates. [kamin+reddy] discuss both approaches to objects and show they are equivalent, although [reddy88] asserts that the closure model is more abstract.

Obliq applications locate foreign objects through calls to very simple name servers. A name server is a process with a distinguished IP address. The protocol between the application consists of four calls:

1. `net_export(``obj´´,` *NameServer*`, obj1)`, where "obj" is a string identifying the object, *NameServer* is a string with the IP address of the name server, and `obj1` is a reference to the object. `Obj1` must be an Obliq object, as opposed to a closure.

2. `net_import(``obj´´,` *NameServer*) returns the location of `obj1`.

3. `net_exportEngine(``Engine1@Site1´´,` *NameServer*`, arg)`, exports an execution engine. An execution engine receives single-parameter Obliq closures and executes them, passing in arg as the single parameter. The argument can serve as the access point to the local environment.

4. `net_importEngine(``Engine1@Site1´´,` *NameServer*) imports a reference to an execution engine to which closures can be sent.

Obliq is very similar to Dreme, and either one could be programmed in the other. Certainly the Obliq object system is implementable in Dreme. Where Dreme extends beyond Obliq is in its support for complete mobility, even down at the implementation level. This allows the exploration dynamic topologies, which would not be possible with Obliq. In this sense, Dreme remains closer to Emerald.

2.6.4 Telescript

Telescript, from General Magic, is another interpreted language supporting some kind of mobility, although General Magic has released almost no concrete information about their language. The name is a pun on *postscript*, the ubiquitous printer language; Telescript hopes to fulfil the same function in the network. Telescript is an agent based language; agents reside at particular hosts and move to other hosts when executing the `go` command. When arriving at a host, the agent must access a name server to communicate with other local agents, and use a rendezvous mechanism to communicate with agents at other hosts. General Magic was very concerned with securing in developing Telescript. Agent identities are verified with digital signitures, and each agent travels with a ticket which states the desired capabilities of the agent and the amount of resources which it is authorized to consume. Hosts may reject agents outright, or terminate them when their authorizations terminate. General Magic further claims that interpreted languages are inherently safer than compiled languages.

It is difficult to compare Dreme to an unknown such as Telescript. By explicitly supporting the agent paradigm and concentrating heavily on security issues, the designers may have limited the flexibility of the language. The use of a rendezvous mechanism for remote communication seems to indicate that this will be more complex than it is in Dreme, making it difficult to write a multiuser, distributed application in Telescript. Their concern for security may have led them to an architecture which creates unnecessary overhead when security is less of an issue.

2.6.5 Actors

The actor model was introduced by Carl Hewitt [36] as a paradigm for distributed AI applications. As such, it predates Communicating Sequential Processes [21], although the latter has been far more influential in the development of concurrency in Algol-like languages. Nevertheless, actors have been very influential in the development of object-oriented concurrent languages. Actors support both fine and medium grained parallelism. "The point of actor architectures is not so much to simply conserve computational resources, but rather to provide for their greedy exploitation – in other words, to spread the computation across an extremely large-scale distributed network so that the overall parallel computation time is reduced."[1]

2.6.6 Act

The most complete elaboration of the actor model is found in [1], and the following description is based on it. An actor system is built of a set of autonomous components called actors which sit around waiting for tasks (synonymous with messages in other languages). When an actor receives a task it can:

1. Create some new actors
2. Send tasks to actors it knows of
3. Perform some computation
4. Specify a replacement behavior for itself

Most actors will respond to a task by creating new actors and sending them tasks, but the model does include provisions for a limited number of base actors (integers, reals, etc.) which respond to a task by performing a computation and (optionally) sending back a task. The last step, specifying a replacement behavior, is the only requiredone, in that an actor which does not specify a replacement behavior (even if it is "do the same thing") cannot respond to the next task. On the other hand, there is no specific order in which these steps are to be performed, so once the replacement behavior is specified, the next task can be received and processed, even while work on the previous task continues, thereby increasing the degree of concurrency in the system. Strictly following the model implies that any computation more complex than arithmetic requires creating new actors, implying very actor-intensive systems. Garbage collection of abandoned actors is, of course, essential for any actual implementation.

Formally, an actor is an element $a \epsilon \mathcal{A} = \{(m, b) \mid m \epsilon \mathcal{M}$, the set of all mail addresses, and b$\epsilon\mathcal{B}$, the set of all behaviors$\}$. The local states function, l, of a set of actors is a mapping from the addresses in the system ($M \subseteq \mathcal{M}$) to their behaviors, i.e. $l : \mathcal{M} \rightarrow \mathcal{B}$. Each mail address is unique in the system, in that there will not be more than one actor waiting for a task at a given address. A task is defined as a tuple from the set $t \epsilon \mathcal{T} = \{I \times M \times K\}$, where I is the set of tags, M is the set of addresses, and K is the set of all possible communications, where a communication is simply the information that one actor wishes to send to the other, a command, return value, or whatever. The tuple t is guaranteed to be unique – there is only one task with a given tag. A configuration is a pair $(l, \mathcal{T})$, i.e., the current actors and the outstanding tasks. Both elements are finite. As the set of behaviors is the centerpiece of the model, the behavior of an actor, a, is an element of $B = (I \times \{m\} \times K \rightarrow F_s(\mathcal{T}) \times F_s(\mathcal{A}) \times \mathcal{A})$, or a mapping from tasks to a new configuration, where the first two elements are new tasks and actors, and the final element is the replacement behavior (with the same mail address). Agha also places the following restrictions on new tags and addresses: the tag of the processed task is a prefix of the tags of all newly generated tasks, and also of the mail addresses of newly created actors (since there is the further restriction that no tag or mail address can be the prefix of any other, this implies that tasks disappear from the system once they are accepted by an actor). This final restriction has obvious utility for controlling an application, as all the actors are now organized in a forest of trees branching by task (it is a forest if the application is seen as starting with some initial set of actors, but it would be a tree if the application starts with a single initial actor which receives a startup task). Actors in the same subtree can be located by changing the last element of the address, and the actors can be garbage collected when the task is completed by string matching on the address.

The basic actor model assumes that all messages are eventually delivered, but assumes nondeterminism in the order of arriving messages, a very reasonable assumption since there is no global clock and no guarantee of maximum delay in a sufficiently large network. The guarantee of delivery is an assumption only that the physical network will not lose messages. Actors have no direct access to mail addresses, but come into being with a finite set of acquaintances and create more acquaintances. Since they only communicate with members of these two sets, all messages go to fixed addresses. In addition, since all actors, during the course of processing a message, must specify a replacement behavior, there will eventually be an actor at that address waiting to receive the message (garbage collection is not part of the underlying semantics, so a garbage collector would need to prove that an actor will never get another message). Note that this, if taken seriously, severely restricts the body of an actor. For example, loops and iteration must be implemented by communication between actors, since infinite loops could prevent the specification of a new behavior (although an infinite loop among actors is fine). For example, recursion is performed by an actor sending a message to itself. In a certain sense, each actor functions as a single line of code in the actor program. As a result of this, mail address generation must be part of the run-time system and addresses must be treated like atomic objects inside the language (even though they might be very large atoms), or else a task could be send to a non-existent location. This is somewhat in contrast to the Lisp world, from which actors originated, in which almost everything is manipulable. Of course any given actor language could provide operations on mail addresses, although at the loss of some theoretical purity.

[1] provides two bare bones actor languages, SAL (Simple Actor Language) and Act. SAL has algol-like syntax and Act has lisp-like syntax. A program in SAL is composed of a series of behavior definitions, *let* statements, and *send* statements. A behavior definition is of the form:

```
def <behavior name> (<acquaintance list >) [<communication list>]
    <command>
end def
```

where <*acquaintance list*> gives the actors which an actor with this behavior knows when it starts, <*communication list*> includes the kind of communication that is sent, as well as the parameters (this is generally in a case statement of the form:

```
case operation of
     op1: (<parameter list>)
     op2: (<parameter list>)
     ...
end case
```

providing a way of interpreting the parameters based on the first element of the list). The command is a non-empty sequence of *if-then-else*, *become*, *let*, *send*, or *def* statements. The *become* statement (including acquaintance list) determines how the actor will act on its next communication, and the *send* statement send a communication to some acquaintance. The *let* command, whose form is:

`let` *x* `= new actor(`*p1*`,` *p2*`, ...)  [and let ...]*` { <*command*>}

creates a series of new actors which can then be used within the inner scope. Act is essentially the same, but with a Lisp style syntax. In both languages, actors are referred to by name. An actor cannot infer the existence of another actor simply from what it knows from its own subset.

Although computationally sufficient (with the addition of operations on base objects, such as integers, etc.), these languages are not easy to program with. Several more sophisticated constructs are added for the sake of ease of use. These include:

1. the use an intermediary object (called a customer) for receiving return communications from an actor in the context of a *let* statement. Here, we use the construct:

   <*statement list 1*> `let` *future* `= (call` *server*`[`*p1*, *p2*`])` {<*statement list 2*>} <*statement list 3*>

   In this case, future is a hidden parameter in the call to server. Server will use a reply statement to send its value back to future, at which time <*statement list 2*> will be executed. In the meantime <*statement list 1*> and <*statement list 3*> can execute concurrently. Creating an object to perform a duty like this is also called *delegation*.

2. insensitive actors are rather the opposite. They enter a wait state in which they buffer all communications while they wait for a reply which will tell them how to process future communications. This can be done by using a customer as above, and then switching to a behavior in which the actor puts all communications on a queue (which is done by creating a list of actors, each holding a communication and waiting to hand it back with a reply) until it receives a message from the customer, at which time it switches to the appropriate behavior to read back the queue.

3. sequential composition, which is done using semicolons, i.e.,

   {<*command list 1*>};{<*command list2*>}.

   <*Command list 1*> is performed before <*command list 2*>.

4. delayed evaluation of infinite structures can be handled using *delay* and *force* commands. *Delay* essentially creates an actor but does not actually send it the message to evaluate the rest of the structure. *Force* causes it to evaluate the next part of the structure (since it should then hit another *delay* command at some point, stopping further evaluation).

2.6.7 ABCL/1

A more "heavy duty" actor language is provided by ABCL/1 (Actor Based Concurrent Language) [62, 55]. ACBL/1 contains a number of modifications to the original model to ease actor programming and to "serve as an executable language for ... modeling and designing various parallel and/or real time systems such as operating systems, office information systems, ... factory automation systems, ... rapid prototyping ... [and] the AI fields ...." As with many other Object Oriented languages, this means restricting message passing to heavier weight objects. "*inter*-object message passing is entirely based on the underlying object oriented computation model, but the representation of the behavior (script) of an object may contain conventional *applicative* and *imperative* features .... Control structures (such as *if-then-else* and looping) used in the description of the behavior of an object are not necessarily based upon message passing."[62]

Objects can be in one of three states: active, which means in the process of running its script; waiting, which means blocked on the receipt of a message (which may be any in a set, as in a select statement) before continuing a script; or dormant, which means the script is terminated, and the object is waiting for a new message to restart the script. In the raw actor model, waiting and dormant simply represent different behaviors. Objects may have local memory, in which case they are called *serialized*, or are without, or *unserialized* (in the raw model, local variables are passed with the *become* statement).

There are two kinds of message passing in ABCL/1. An ordinary message, M, sent to an object, T, is serviced according to T's state. If T is dormant, then M is checked to see if it is accepted by T's script. If so, it is processed, otherwise it is discarded. If T is waiting, then M is checked against the current conditions that T is waiting for, and accepted if it matches, otherwise it is placed on the ordinary message queue. If T is active, M is placed on the waiting queue. An express message, E, bypasses this. If T is processing an ordinary message, M, in either active or waiting modes, then M is suspended and E processed. If T is already processing an express message, then E is placed on the express queue. Express messages function much like interrupts. (It is possible, however, to designate certain sections of code as atomic.)

Each kind of message comes in three flavors: *past* type message passing, in which A sends a message to B and A continues with its own computation (written

[B <= A] in ordinary mode and [B <<= A] in express mode; *now* type, in which A sends to B and blocks waiting for B's reply ([B <== A], and [B <<== A], respectively); and *future* mode, in which A additionally passes a result object to B into which the result will be put for A to access at some point in the future ([B <= A $ x] and [B <<= A $ x]). In the last type, x can be queried to see if it can be accessed, and any number of results can be picked off, one at a time (a premature access suspends the reader). Naturally, both of the latter types can be reduced to *past* type messages.

ABCL/1 does not have a *become* statement, but this can be mimicked by having the object move from wait state to wait state, where each desired behavior corresponds to a different wait state. The top level messages designated by the script are never returned to, and would therefore only be available to express messages. They would serve as an interrupt table.

Objects are defined in ABCL/1 with the following syntax:

```
[object object -name
         (state local state declarations )
               (script
                      (=> message-pattern1 where constraint1
                      actions1)
                      (=> message-pattern2 where constraint2
                      actions2)
         ...
         )]
```

Scripts can recursively contain object definitions, so it is possible to create constructor objects, whose purpose is to create new objects of a given type, like constructors in class-based languages like C++.

ABCL/1 is a continually evolving experimental language. In the version discussed here it has been used for numerous programs, including English parsers, parallel discrete simulation, robotic control, and some AI work [62]. A distributed version in Lisp running on a network of Sun workstations was developed [8]. Later versions of ABCL/1 have added support for exception handling and reflection [22]. Exceptions are trapped by enclosing some part of a script with a catch-signal construct:

```
(catch-signal
             ... actions ...
      (handler
             (=> signal-pattern
             handler-body)
...
))
```

There are a few ways to raise signals. For instance, various constructs can have time periods attached, such as a message that expects a reply [T ¡== M % 100], which will raise a signal if it is not performed in 100 units, explicitly call `(raise ...)`, encounter a system-defined error condition (i.e., divide-by-zero), or encounter an unacceptable message (i.e., a message sent to an actor for which there is no script). In the last case, the signal is `[:unaccepable M :from sender :to T]`, which is sent either to the sender or to an explicitly stated complaint destination that was included in the original message.

2.6.8 Actors and Dreme

In papers such as [19, 20], it is clear that actor semantics are equivalent to a restatement of denotational sematics. As such, all languages can be seen as variants on actor languages, Dreme included. As a version of Scheme, Dreme is closely connected to actors, as Scheme's original development was partly to concretize some of Hewitt's ideas for actors.

If we look at actors in a more restricted sense, as message passing languages with the particular *become* semantics and a continuation-passing style, then we see that Dreme remains more closely wed to traditional languages, although implementing an actor language would be quite straightforward. Dreme also has some elements which are not found in actor languages, such as mobility, an awareness of the physical topology, and the ability to set policies for the behavior of objects in a particular process.

2.7 Lisp Systems with Mobile Objects

Parallel Lisp systems date back to early Actor work. Multilisp [52], for example, is an early parallel implementation of Scheme which introduces two important constructs, the *future* and the *pcall*. These systems have tended to be on shared memory multiprocessors or multithreaded on a single processor. Here we will look at some systems which assume distributed memory and even some degree of mobility.

2.7.1 Concurrent Scheme

Concurrent Scheme (CS)[31, 56] is a dialect of Scheme developed for the Mayfly [30], a distributed memory multiprocessor, but which also runs on networks of workstations. CS borrows the concept of a *domain* from Hybrid[42]. Domains are analogous to monitors. Only one thread can be active in a domain at a time, providing protection against concurrent access, but also elevating threads and requiring a set of primitives to handle them. All computation takes place within some domain, providing mutual exclusion for data access, but requiring other threads to be queued. Inside a domain, the primary form of encapsulation is the closure, which is treated like an object. The entirety of a closure should reside within a single domain; global variables are discouraged. Data is copeid when it moves from one domain to another. Only domains, threads, placeholders and closures, which must enclose state, are passed by reference.

Operations on domains include:

- `make-domain`, which optionally takes a network node and heap size.
- `(apply-within-domain <proc> <arglist> . <domain>)` invokes `<proc>` on `<arglist>` in `<domain>` and returns the result. The domain parameter is optional if the procedure is a closure. The current domain remains occupied, so this is similar to RPC.
- `(delegate <proc> <arglist> . <domain>)` acts as above, except the current domain is vacated while the thread is migrated. When the result arrives, the thread of the continuation must wait on a queue if the domain is occupied.

CS also includes primitives to manipulate threads, as they are critical to the domain approach. The language uses *placeholders* as an analogue to futures with the same semantics.

The fundamental difference between CS and Dreme is in the former's insistence on tying all closures and mutable data to particular locations on the network. This is partly intended as a means of single threading access to mutable data, but it is not clear why a construct such as *monitors* is not sufficient. Semantically, CS is a restricted subset of Dreme where closures are automatically pinned upon creation at its site of origin, and all other data is copied rather than moved when sent from node to node.

2.7.2 ICSLAS

ICSLAS[49], a successor to the language described in [47], is a distributed Scheme dialect with similar goals to Dreme, but with a very different emphasis. At the lowest level, ICSLAS depends on an object-oriented system using generic dispatch, similar to CLOS [32] and the developer's own Meroon[48].

In such a system, grosso modo, objects are records and are completely separated from the code that manipulates them, which is hidden in generic functions. Each generic function contains one or more method bodies; each method body has a different signature and can be placed in a lattice accordingly. When a generic function is invoked, a dispatch mechanism examines the types of the parameters and chooses the most appropriate method body and applies the method to the parameters.

The natural means to accomplish distribution in such a system, and the approach taken by ICSLAS, is to make the generic functions constant on all nodes and to have the objects migrate to them when the function is invoked. In ICSLAS these objects are very small. The source code is converted to a network of fine-grained objects representing the executable code in a CPS format. When a generic function on some node requires one of these objects it is brought over, and *husks* (an empty chunk of memory that a remote object can later overwrite) created for its neighbors. When the object has been used and the next one required, that is brought over to replace its husk.

ICSLAS and Dreme disagree in two pragmatic aspects and one philosophical one. Dreme is byte code interpreted so a function or object executing on a single node of the network does not pay a price in performance because it could have been distributed. The representation of the "compiled" program as a network of small objects organized in CPS is very similar to the approach taken in an early version of the Dreme interpreter. This was rejected in favor of the less object-oriented byte-coded alternative uniquely for reasons of performance; a later version might ship platform-independent byte code and convert it to platform-dependent object code on arrival. Also, given that round trip network latencies are relatively fixed, and that processing often maintains some locality of reference (a thread entering a function or closure is likely to spend time there), it is better to send more data over the network together. Dreme uses Scheme's own scoping mechanisms to help determine how large this is.

The more philosophical issue is related to the use of generic functions for implementing an object oriented system in the network. Generic functions and message passing represent two different styles of object oriented system. In a distributed system where objects are developed independently, the use of generic functions can be problematic. Given two objects, a thread, a call to a generic function, and various method bodies scattered around the network, it is not obvious where processing should take place. By contrast, message passing clearly distinguishes the recipient of the message as the next location of the thread of control. In addition, the message passing approach better handles encapsulation, a prerequisite for any security.

2.8 Java

A very recent development which deserves some mention here is Java[16], an object-oriented language developed by Sun Microsystems. Java originally started as a project to extend C++ to support run-time linking, but then evolved into a type-safe interpreted language depending on a fast byte-coded virtual machine (VM). Java's visibility has risen considerably through the recent release of HotJava, a WWW browser which allows applications to be imported from a Web server.

The fundamental aspect of Java is its support for implicit run time linking of new classes. All object code is divided into classes which reside in separate files. A Java program starts through the instantiation of an object of a particular class. As the object runs, it creates other objects from other classes, however the code for those classes does not need to be physically present in the program. Trying to instantiate an object of a non-existent class generates an exception which is handled by a class loader object. The object code for the class is located, imported and linked into the interpreter, and the instantiation attempt is retried. By incorporating an exception handler which can access a network, a Java class can reside anywhere.

This implicit linking has an interesting impact on Java syntax. Because classes must be self-contained units (as they can be called from anywhere), Java programs do not contain global variables. However some of the functionality of multiple scopes can be recovered through the inheritance mechanism. Java supports single inheritance and, following in the tradition of C++, has *static variables* (called static members in C++). These static variables are members which exist once for a class, instead of once for each object. These variable could be used to create a tree of

```
   (define name-server '())

   (set! name-server (cons (cons 'service-name service-function) name-server))

   (let ((service (assoc name-server 'service-name)))
      (if service (service args)))

   (let ((service (assoc (remote-eval server '%name-server)
 'service-name)))
      (if service (service args)))

   (let ((service (remote-exec server
      (lambda () (assoc %name-server 'service-name)))))
      (if service (service args)))
```

Figure 2.9: Minimal Distributed Object

scopes of static variables with each subclass adding its own group, until at the leaves there are objects with traditional members. This would not be equivalent to true nested scoping; in Java there are an unlimited number of objects with private members, but the number of static variables is fixed by the number of classes.

Java differs from Dreme by only supporting run time transport of object code, i.e., the distributed delivery of self-contained applications, while Dreme, by contrast, supports movement of both code and data for the distributed delivery of distributed applications. Implementing Dreme's features in Java would not be an easy task. An interesting alternative, which we are contemplating, would be to port Dreme to use the Java VM.

2.9 A Dreme Example

Having introduced Dreme's extensions to Scheme, we will now give a brief example of a distributed application. We will show three iterations of making a distributed

```
(define add-svc #f)

(define name-server
  (pin (letrec
           ((serv-list ´())
            (add-func
             (lambda (name service)
               (if (and (symbol? name) (function? service))
                   (set! serv-list (cons (cons name service) serv-list))))))
         (set! add-svc (pin add-func))
         (lambda ()
           (lambda (name . args)
             (if (symbol? name)
                 (let ((service (assoc name serv-list)))
                   (if service (apply service args)
                       (throw ´not-available)))
                 (throw ´invalid-name)))))))

(define name-server (remote-eval server ´(%name-server)))
```

Figure 2.10: Minimal Server Object 2

```
(define add-svc #f)

(define name-server
  (pin (letrec ((serv-list ´())
                (add-func
                 (lambda (name service)
                   (if (and (symbol? name) (function? service))
                   (set! serv-list (cons (cons name service)
                                         serv-list))))))
         (set! add-svc add-func)
         (lambda ()
           (let ((client
                  (lambda (name . args)
                    (if (symbol? name)
                        (let ((service (assoc name serv-list)))
                          (if service (apply service args)
                              (throw ´not-available)))
                        (throw ´invalid-name)))))
             (if (eq? (assoc security-level ??sender??) ´trusted)
                 client
                 (fix client)))))))

(define name-server (remote-eval server ´(%name-server)))
```

Figure 2.11: Minimal Server Object

```
(define add-svc #f)
(define del-svc #f)

(define name-server
  (pin (letrec ((serv-list ´())
                (client-list ´())
                (add-func
                 (lambda (name service)
                   (if (and (symbol? name) (function? service))
                   (set! serv-list (cons (cons name service)
                                         serv-list)))))
                (del-func
                 (lambda (name)
                   (par-map (lambda (x) (x ´delete name)) serv-list)
                   (removeq serv-list name)))
         (set! add-svc add-func)
         (set! del-svc del-func)
         (lambda ()
           (let* ((letrec
                      ((priv-list ´())
                       (manip-list
                        (monitor (action name)
                          (cond
                           ((eq? action ´add)
                            (set! priv-list (cons name priv-list)))
                           ((eq? action ´good?)
                            (if (%assoc priv-list name)
                                (cdr (assoc serv-list name)) #f))
                           ((eq? action ´delete)
                            (removeq priv-list name))))))
                    (set! client-list (cons manip-list client-list))
                    (lambda (name . args)
                      (let ((val (manip-list ´good? name)))
                        (if val (apply val args)))))))))))
```

Figure 2.12: Minimal Server Object

name server in Dreme, each more complex than the previous.

Figure 2.9 gives a minimal, and not very elegant, version without any lexical scoping. The `define` creates a list in a Process we will call *server* and gives it the name `name-server`. `Name-server` can be manipulated locally like any ordinary Scheme object. For example, we can add services using `set!`. Other Dreme Processes can access `name-server` using the `remote-eval` macro. `Remote-eval` wraps the quoted expression in a thunk and sends it to the appropriate Process, which might be the local one, to be executed (the call `(remote-eval server '%name-server)` is equivalent to `(remote-exec server (lambda () %name-server))`). The expression in line 8 would execute correctly both locally and remotely. The call to `remote-eval` evaluates to the object `name-server` refers to, which is the list of `(name . service)` pairs. If the call is actually remote, then the mobile portion of the list will migrate to the remote Process. Were there several remote Processes accessing the list concurrently, the elements of the list would be shuttling between them as they iterate down its length (the identifier `name-server` would remain at the same location, but the object it refers to would be constantly moving). This implementation of a name server is not particularly efficient, and it is not particularly safe, as access to the server is completely unrestricted. The expression starting on line 11 is more efficient, as it ensures that the service is looked up on the server. There may be several clients looking at once, but they will all be looking in the same place. Nevertheless, the list of servers remains completely exposed.

Figure 2.10 provides a slightly more sophisticated server in which the list is encapsulated by two functions, `add-svc` and `name-server`. The first one adds services to the list, and the second one searches for them on behalf of clients. A client gets access to this second function by calling `name-server` in the server prcess. In this version, each client receives its own copy of `name-server`. In general, though, there are a number of ways in which the server Process and various client Processes can interact, based on the degree of trust between the two parties or other considerations, while maintaining the same client interface. 2.11 provides two basic interfaces, one in which the client is shipped over the network and one in which only a pointer is sent, based on the identity of the caller. As written, the list of services will be spread among the clients if there are multiple simultaneous accesses. This shuttling can be eliminated by `pin`ning the cons cells that form the list; since the list would be immobile, the list woudl be searched at the server.

The last version 2.12 is far more complex and has the server updating the clients as services arrive and disappear. This breaks the "normal" client server architecture in that the interaction is not all initiatted by the server. This kind of interaction, however allows the client to, for example, display an up-to-date list of services. The `assoc` operation on the private list is guaranteed to occur locally, where as the `assoc` operation on the shared list will occur at the server site. The private list and the `manip-list` function which accesses it are guaranteed to be local as they are only accessible through the client closure returned with the invocation. It also includes some concurrency control between the parties – services are not removed at the server until they have been removed from the clients. The clients are informed with a parallel map `par-map`. This contacts all remote clients asynchronously, but does not terminate until all the calls are completed (since it must return a list to the caller). The private lists at the client sites are guarded by monitors so that the server cannot delete services while clients are looking for them.

Chapter 3

Previous Experiments

The conceptual seeds of Dreme grew out of work on computer-integrated manufacturing (CIM) and on the Pad next-generation graphical user interface. Although somewhat brief, my flirtation with the former led to the use of mobility and lexical scoping in Dreme; while the latter work led me to reduce the role of operating system processes in Dreme and to eliminate them from the GUI, where they do not present a relevant user interface metaphor (unless the application is directly concerned with managing operating system components). Both led me to support a uniform distributed object space. Another area of interest, although I didn't do any experimentation, was managing interactions with multiple online services.

The common thread running through all of these was attention to distributed programs linking together already existing objects and services in the network, a opposed to a "stand alone" parallel or distributed program which functions as if it were in an empty network.

3.1 From Programming to Manufacturing and Back Again

My first effort in CIM was an object-oriented graphical front end for the NYU open architectured machine tool. In this innovative device, all the components of the machine tool were placed on an industry standard VME bus, controlled by a real-time UNIX variant, and addressed over a TCP/IP network. Any workstation on the network could access the machine tool, either to send it machining commands, or to receive back information.

For this machine, I developed a GUI in X to allow the user to manipulate a block, describe various machining operations to be performed on it, and generate the appropriate commands to be sent over the network to the machine tool.

After this initial success, there were two dimensions to expanding the interface: increase the depth of the interface in communicating with this particular machine, or start considering more complex objects (such as cars, etc.) which require assembling many components and controlling many machines. It was the latter which most influenced my further work.

From a high level, there were three aspects to building complex devices which were immediately evident:

1. The system would need to run on a network. This was a problem, in that the commonly available tools for network programming, such as RPC, were cumbersome and not designed for a dynamically evolving application. In addition, any concern for the factory network was a conceptual nuisance for a designer, as it would have little, if any, relevance to the parts to be manufactured, and would impede portability of the design. It had to be possible to address the network when necessary, but otherwise ignore it.

2. Any design for manufacturing a set of objects that relied on particular machines being available at any given time would eventually fail; machines would break, or priorities and rush orders would alter the set of available machines. Therefore it was imperative that any code to perform an action needed to be dynamically migrateable to some other machine without affecting the rest of the manufacturing process. At the same time, the network could not be seen as just a massively-parallel processor – a factory would present a very heterogenous network, and certain operations required certain resources.

3. Manufacturing an object such as a car takes place in several locations and reliance on a central computer to continuously orchestrate everything creates both a single point-of-failure and a potential network bottleneck. If we consider manufacturing as a decentralized process, then different parts of the "program" need to reference each other. The location of parts of this distributed program might change constantly, as machines fail and schedules change; the underlying system must ensure that communications always arrive at the correct place in a transparent manner, regardless of the migratory behavior of sender and receiver. This requirement leads directly to the concept of distributed lexical scoping.

Although I did not continue to develop the CIM prototype, I brought these ideas back to the more general field of distributed programming. Each of these plays a role in the development of Dreme.

3.2 Backends for Pad

Pad is a graphical user interface developed at NYU using a "whiteboard" metaphor. A Pad surface is an infinite plane of infinite resolution. Since the resolution is infinite, it is always possible to place more information in a given area of the plane by "writing smaller" and to retrieve it later by "looking closer" (*zooming*). Pad also contains mechanisms, similar to *hyperlinks*, for refencing an area of a surface from another, so either the same object is seen at more than one location, or there is a *portal* with a view on other location on the surface, so that on looking closer, one would find oneself at the other location.

Upon seeing Pad, I was immediately struck by its appropriateness as a GUI for truly distributed applications. It made me realize that the "traditional" windowing metaphor, as exemplified by X, was a stumbling block to building a system that could adequately handle meaningful access to remote resources.

In a typical windowing system, the available screen space is divided by tiny rectangles. Each rectangle is connected to a particular process and all events associated with that window go to the particular process. Processes do not ordinarily communicate except through "cut and paste" of text strings. From this perspective, X appears as a visual metaphor of the underlying Unix operating system: the screen is the CPU, which is multiplexed by the processes competing for "turf"; the user is the scheduler and moves the mouse to designate the "current" task; finally, "cut and paste" replaces pipes.

This metaphor provides little assistance to users who want to have several online services cooperate with one another in fulfillment of a particular task. It is possible for each to display its own particular interface, but then they all exist in isolation. Interaction is being forced into the metaphor of the operating system, instead of following metaphors more appropriate to the domain. The windowless architecture of Pad opened the possibility to a distributed user interface in which a seamless space of interacting objects could be presented visually, even though behind that the objects resided in different processes on different network nodes.

An example would be a distributed CAD system where two components could be connected on the screen, but the actual implementation code was running in two different servers and being controlled locally by a third process. In an ideal implementation, manipulating the distributed CAD representations would be indistinguishable from manipulating non-distributed one. If these two components were actually manufactured by two separate companies, then Pad, combined with the proper backend, could provide an appropriate substrate for commerce on the Internet.

Providing that back end was obviously more than an afterthought. If the goal was distributed commercial services, then there was the implication of a general distributed application architecture which could run over a public network. This brought with it a whole host of issues; in the fall of 1990 I started to explore them.

I built two backends for Pad to support distributed clients with a seamless interface before stopping to develop Dreme. The first one used a single Pad and multiple clients; the second allowed multiple Pads and multiple clients.

3.2.1 The Pad Factory

The Pad factory attempted to convey a unified interface to a distributed system. In the foreground was a single Pad, in the background were several different processes representing machine tools, robots, and parts in a factory. The GUI, although quite primitive, was divided into a “factory floor” section, which directly represented the objects; and a menu-style section, which provided another view of the same system. One could interact with the machines and robots through either path; the Pad’s redirection features meant that the same command widgets were accessed. For each class of object (whether robot, machine, or part) where a graphical component (such as pull-down menus) were shared, only one copy would appear on the Pad; the path from an event to that component would determine which background object was notified.

At the software level, the system was divided into clients and Pads sending messages to each other. In the one direction, these were commands to change the graphical appearance of the Pad; in the other these were user events. To minimize potential processing delays in each direction, each workstation running a client or Pad also had a post-office daemon which maintained mailboxes for local processes. When a Pad or client wished to communicate, it would send the message to the post office on the receiver's machine. This would then, in turn, signal the receiver that a message was ready. The receiver could then pick up the message when it was convenient to do so. Neither Pad nor client would be blocked due to a slow partner; the post-office processes remained potential bottlenecks, but their function was very simple and they were therefore rather quick.

Since there was a desire to drop the turf-based windowing paradigm, each graphic object on a Pad had its own unique identifier and was addressable by any client on the network. Each Pad object had a list of possible events that could occur to it. Clients would announce the events which interested them on a per-object basis; when an event occurred, all interested clients would be notified, whether the creator of the object or not. Likewise, any client could send commands to any object, or create new objects in any Pad. On the factory prototype, most of the object types were fixed and shared among the client, so they could be loaded at run-time. As clients connected with the Pad, they would query the object data base to locate the objects they needed and create any that were not present.

Although the factory used only one Pad, the model had no such restriction. The intention was to expand to having several Pads displaying all or part of the factory. Since each user might have different rights and responsibilities, each user might see a different subset of the objects in the factory and even view different representations of them. Security questions, such as access rights for clients to different objects, would have been considered. This would have led to an interesting problems both for the back-end and for the front-end, but this work was put aside when the version of Pad that had been used was abandoned by the project.

This prototype demonstrated the possibility of an "open architectured" user interface for distributed systems. At the same time, the possibility of expanding to multiple Pads demonstrated the need to move to multi-threaded clients. Both clients and Pads functioned around event loops. In the case of the Pad, this was not a problem, as no process state was needed between events. Clients, however, might be required to handle events from multiple Pads simultaneously. This may have also been at the root of my decision to find a platform independent solution to the user interface; the distributed back-end was entirely dependent on the use of a particular version of Pad. Although I still think that a version of Pad will be the best front end for Dreme, Dreme is not dependent on any particular user interface.

3.2.2 Multiple Pads

The second distributed Pad backend was written in C++ and used remote method invocation to create a shared whiteboard. C++ classes were created for elements of Pad, each with a corresponding proxy class, so that methods could be invoked remotely. In this scenario, there were several participating Pads. One of the Pads acted as server. Whenever an event happened at a particular Pad, the event would be relayed to the server, which would notify all the participants.

The obvious drawback of this approach was the existence of the centralized server as a bottleneck for the whole system. Another problem was the lack of an automatic means for generating proxy classes, which meant that all the marshalling and demarshaling code needed to be written by hand. Solutions to both of these existed for distributed C++ objects, the former with more sophisticated resource sharing techniques, and the latter with a stub generator or CORBA implementation.

The fundamental problem of a C++, or other approach with a compiled executable, is the inability to support new classes and procedures at runtime – the structure of all applications is decided at compile time. This is a very rigid structure for supporting a network full of objects and services. Either all objects fit a very rigid framework, with an inevitable loss of flexibility, or there are services we cannot access. Neither is acceptible. Another drawback of this approach is a lack of flexibility in load balancing the work of the server. Since the server appears as just an interface to the client process, the client process cannot assist the server in more than the most cursory fashion. If the server can provide some information to the client on how data is to be handled, however, the client has the ability of offloading much of the work that the server needs to do.

3.3 Conclusions

Once I had done the work described here, I had several criteria for the system I wished to build. The next breakthrough was realizing that Scheme provided a framework which I could expand naturally to cover almost all of them. The remaining pieces, involving the user interface, are handled by the SGML-based user interface.

Chapter 4

A Multithreaded User Interface

The Dreme distributed run-time system allows users to communicate with objects all over the network and allows objects to come to the users as necessary, but so far we have not discussed any aspects of the user interface. While it might be possible to map all Dreme communication into a command-line interface, the richness of possible actions and the variety of objects that can be in the environment calls for a graphical interface. The Dreme architecture makes three strenuous requirements for this:

1. The interface must be multithreaded. We have already discussed the underlying reason that GUIs should be multithreaded, but Dreme objects able to spin off threads easily, so this is even more significant in the case of Dreme.

2. The interface must not require any particular underlying architecture. Since Dreme objects may find themselves on any node in the network, they must be able to display their interface regardless of whether the host is a Unix, MacIntosh, MS Windows, or other system. For the sake of developers, it should be possible to develop a single interface for an object that runs on all systems.

3. The interface should make it easy for users to combine objects. Most current interfaces, even multithreaded ones like X (in the sense that it can display windows for multiple clients), are essentially "turf-oriented", meaning that the available real estate is divided into sections, each owned by a single process. Processes rarely communicate, and when they do they exchange "dead" information — such as character strings. In Dreme all objects are individually alive and able to communicatebe, and the interface should reflect that.

We have chosen to develop a three level architecture for GUI based applications which will be outlined in the first two parts of the remainder of this chapter. In order to finish the chapter with a fairly complete example, we will first treat the visual appearance of the application. Our approach equates user interfaces with documents, in the sense of SGML, and describe how to create platform independent user interfaces through the concept of a "logical" interface. We will then briefly show how this approach can be combined with the World Wide Web (WWW) to support a general purpose intelligent WWW browser of functionality between that of Dreme and current browsers, one that can not only display a wide variety of hypermedia information, but also actively manipulate it. After that we will describe the mechanisms used for concurrency control and linking the application to the GUI in a multi-threaded distributed application, based on the techniques laid out in chapter 1. Using the example of a game of bridge, we will show how appearance and behavior are related.

4.1 The Logical GUI - on beyond X

The previous section presented a new style of GUI programming based on closures and continuations, which let the application be written in a natural style as opposed to a continuation passing one. In this section we will show how we have separated the application user interface from dependence on a particular platform and widget toolkit by allowing the UI to be specified in a logical manner and only later mapping that into a particular widget set. This allows a mobile object to present its interface to users on a variety of platforms. This is achieved through extensive use of the Standard Generalized Markup Language (SGML), an international standard for specifying markup languages for document types. The same techniques can be used to browse and manipulate network-linked hypermedia documents, such as the HTML documents of the World Wide Web, but in a wide variety of formats, significantly beyond the current abilities of the Web.

4.1.1 A Brief History and Explanation of SGML

The Standard Generalized Markup Language (SGML) grew out of an IBM project in the early 1970's by Charles Goldfarb for storage and printing of legal documents. That effort was spearheaded by Charles Goldfarb, the editor of ISO 8879 (SGML), and resulted in the development of GML.

Although SGML is ordinarily used for document processing, SGML difers from ordinary word processing systems through its distinction between logical markup and procedural markup. In order to display a plaintext document with the appropriate fonts and formatting, word processing system generally insert control codes, called markup, among the characters of the text to describe various substrings. Procedural markup inserts codes that tell the formatting program exactly how these strings should appear, whether on the string or on the printed page. Logical markup, on the other hand, identifies substrings as syntactic elements of a document, and leaves the direct mapping to formatting codes to a later stage in document processing.

Logical markup provides much greater flexibility than procedural markup for document processing because, when done properly, it reflects the structure of the document. This makes it possible to search a document for particular elements, to disassemble it into its constituents, to store the parts in a database, and to create new documents automatically. This is almost impossible with procedural markup, as the same visual appearance may be given to different semantic elements of the logical document.

Logical markup, as exemplified by SGML, has the additional benefit of strongly separating the definition of the markup which represents the syntax of the document, from the manipulation of the document, which represents its semantics. This allows a variety of different semantics, eg. a presentation semantics and a database storage semantics, to be used for the same class of documents.

4.1.2 The Elements of SGML

This is not the place for an exhaustive description of SGML, but we will provide a sufficient description for the reader to follow the examples.

SGML is a system for specifying a class of context sensitive grammars. An SGML application consists of four parts:

1. A *document type definition* (DTD) which describes the grammar to be used.

2. A *document*, which is a string in that language

3. A *parser*, which takes the first two elements as input and outputs the abstract syntax tree of the document.

4. An *application program* which uses the parsed document in some way.

The DTD The DTD consists of a series of records. We will need four of them:

1. *Elements* define the production rules of the language. An element definition consists of six parts:

   (a) The keyword element.

   (b) The name of the element. The substring of the document is bracketed by an opening tag, of the form `<element-name>`, and a closing tag, of the form `</element-name>`.

   (c) Whether the opening tag is optional, or is to be inferred by the parser. This is either an "O", for optional, or a "-", if it is not. The parser may reject this if it is not always possible to infer where the tag open should occur.

   (d) Whether the closing tag is optional. The same conditions apply as above.

   (e) The content model of the element. This is the right-hand side of a production rule in a traditional context-free grammar. It consists of a list of terminals (the plaintext of the document) and non-terminals (other elements). These can be grouped by parentheses and separated by the following: * for zero or more repetitions, + for one or more repetitions, ? means the element is optional, — means one (but not both) of the elements on either side must occur, and a comma (,) means both must occur in that order.

   (f) Inclusions and exclusions, which we won't use, but mention for completenes.

   For example, the following element definition:

```
<!ELEMENT (set | append | prepend) O -
                       (path+, (tag-text | attribute | comp-list*))>
```

defines three elements (set, append, and prepend) with the same structure. Each of these consists of one or more occurences of a `path` element followed by either a single `tag-text` element, a single `attribute` element, or zero or more `comp-list` elements. The "O" after `prepend` indicates that start tag for any of the defined elements is optional, but the closing tag is not, so `<path><tag-text></set>` is legal, but `<set><path><tag-text>` is not.

2. *Entities* are references to chunks of information to be substituted elsewhere in a DTD or document. Given an entity declaration such as:

   ```
   <!ENTITY www ``World Wide Web´´>
   ```

   The character sequence `&www;` in a document would be replaced by `World Wide Web` by the parser. A slightly different form of entity is used within a DTD.

3. Attribute lists represent attributes which can be added to an element. These occur as sequences of *attribute name* `=` *attribute value*, or just *attribute value* within the start tag. An attribute definition, such as:

   ```
   <!ATTLIST set attr1 (val1 | val2) val2
                 attr2 (val3 | val4) #IMPLIED
                 attr3 CDATA #REQUIRED
                 attr4 ID #REQUIRED
                 attr5 IDREF #IMPLIED>
   ```

defines five attributes for the element `set`. The first, `attr1`, has a value of either `val1` or `val2`. If it is not present in the document, then the parser will substitute `val2` as the default value. `Attr2` does not have a default value; `#IMPLIED` indicates to the parser that the application can determine the appropriate value. `Attr3` is composed of character data and is required. `Attr4` is a required unique character string which identifies this tag, and `attr5` is an optional string referring to the identifier of some other tag. A start tag for `set` might look like:

```
<set attr1 = val1 attr3 = ``This is required´´ attr4=just-me>
```

4. Short reference maps (shortrefs) specify short character strings, such as parentheses, which can be used in a document in place of longer open and close tags. When a short reference is seen by the parser, the longer tag is substituted. The rules for creating short references are more complex than for entities, elements, and attributes. In the examples given below, we will use shortrefs to give a more concise "programming language" look to some documents.

The Document, Parser, and Application An SGML document is just a string of text with markup embedded at various points. The markup may be more or less intrusive, depending on whether tags may be dropped or substituted for with shortrefs. Figure 4.3, below, provides an example with a large ratio of markup to text. The document must be grammatically correct, given the DTD. This is determined by the parser, which, given a DTD and a document, must determine if there is a correct parse for the document and present it to the application. The application then communicates with the rest of the environment, based on the contents of the document.

4.1.3 An Evolving Perspective on SGML

Our use of SGML was originally rather limited. We developed a small language, called the User Interface Definition Language (UIDL), to capture the shared functionality underlying most UI toolkits. Each node/user had a UIMS understanding UIDL. Objects passed their UIs to the UIMS. It generated the appropriate UI objects and returned a handle to the originating object. Afterwards, the object communicated directly with its interface, occasionally passing additional UIDL specifications to the UIMS when new interface elements were needed. SGML was a convenient mechanism for specifying UIDL.

We then considered merging UIDL and the the World Wide Web's (WWW) HyperText Markup Language (HTML)[3], another SGML-based UI language. A merger of the two would have had the desireable outcome of making made all Web clients become Dreme clients as well. Nevertheless, HTML, and by extension UIDL, or any single document type, was deemed too limiting. An SGML document is a sophisticated record type, much of whose information is contained in the markup and is lost when the document is converted to some single-purpose document type. Display, the reason for converting to HTML, is only one of many potential applications. HTML is of little use to programs, which require the explicit structural information of the original document. HTML's utility decreases as client intelligence increases, particularly in the presence of mobility. If code and data can migrate transparently around the network, documents can be transported with their original markup; any additional information, such as the DTD or the display semantics, can be retrieved as necessary. Therefore we chose to support SGML *in toto*.

From this new perspective, a UI is just an SGML document of some type, manipulable by many applications, one of which may be the UIMS. Each UI must supply two additional elements: the DTD, and a default display semantics. These may be shared with other applications, but must be available for the recipient to treat the document as more than a bit stream. At the same time, documents can become applications by being read and interpreted by waiting local software having different semantics than just display, treating them as programs. Our approach attempts to support all these by considering them as aspects of the mobility of information.

4.1.4 Documents are programs: linking the document to the application

The SGML standard, and associated standards such as DSSSL [24] and HyTime [25], are explicitly application independent; no constraints are made on the structure of the application. Our applications, however, need to be as transportable as documents. We also anticipate a proliferation of document types, so applications should be easy to construct. For this reason we impose a common architecture on our SGML applications, one which is succinct, portable, and controllable by the client. Although the power of any particular application may vary, the architecture can support Turing-complete computation, so there is no loss of expressive power *vis-a-vis* other architectures.

We consider a DTD as defining a language, and the marked-up document as a program, with tags representing commands or other sytactic elements. Unlike most programming languages, these languages may have several semantics, depending on the purposes to which they are put. SGML applications are interpreters, one for each semantics to be applied to the underlying language.

4.1.5 The structure of an interpreter

Our interpreters require three components (this is not intended as a definitive statement about interpreters, but only our particular needs):

1. A *virtual machine* (VM) describing the world on which the program operates. This is a set of operations and a set of classes delineating the objects which can exist in this world. The goal of an interpreter is to map from language statements to VM operations. The VM can be seen as providing a semantic domain for the interpreter.

2. A (potentially empty) set of objects existing when program interpretation starts. These objects represent the start state of the VM.

3. A mapping from operations in the programming language to one or more VM operations. These operations may add new objects to the current state, or change the state of existing objects. The mapping provides the semantic actions for the interpreter. The most complex part of the mapping is the mechanism for locating objects in the current environment, which includes both the VM state and the nodes of the document,

VM operations represent the interface to the local host and must be supplied locally; otherwise the intepreter is self contained. By controlling the VM operations, the local host retains control over the interpreter. The same program may be run in two different environments, depending on circumstances.

An interpreter is also a program, written as a document. Our interpreters are SGML documents, transmittable around the Internet, and just as manipulable as the documents they are intended to manipulate. Figure 4.1 provides a DTD for simple interpreters that construct UIs. More specifically, the DTD provides

```
<!ENTITY % tkw1 "toplevel | button | checkbutton | radiobutton ">
<!ENTITY % tkw2 " listbox | label | entry | radiobuttonlist | text ">
<!ENTITY % tkw3 " message | frame |menu | menubutton | menubar">
<!ENTITY % tkwidgets "%tkw1; | %tkw2; | %tkw3;">

<!ELEMENT tkmap O O (name, widget-list)>
<!ELEMENT widget-list O O (comp-list, tag-list)>
<!ELEMENT comp-list O O ((cname | tag-text | attribute), widget-def)*>
<!ELEMENT cname - O (#PCDATA)>
<!ELEMENT name O - (#PCDATA)>
<!ELEMENT widget-def O O ((%tkwidgets;), comp-list?, bindings?)>
<!ELEMENT (%tkwidgets) - O EMPTY>
<!ATTLIST (%tkwidgets) initstr CDATA #IMPLIED
                       packstr CDATA #IMPLIED
                       textstr CDATA #IMPLIED>
<!ELEMENT bindings - - (binding+)>
<!ELEMENT binding - O (event-name, bind-spec)>
<!ELEMENT event-name - O RCDATA>
<!ELEMENT bind-spec - - RCDATA>
<!ENTITY % cmd-list " set | append | prepend ">
<!ENTITY % tagging "(tag, (%cmd-list;)*, tag-list?, (%cmd-list)*)*">
<!ELEMENT tag - O (#PCDATA)>
<!ELEMENT tag-list O - (%tagging;)>
<!ELEMENT (set | append | prepend) - -
                   (path, (tag-text | attribute | comp-list))>
<!ELEMENT path - O (up*, clist)>
<!ELEMENT up - O EMPTY>
<!ELEMENT clist O O (#PCDATA)>
<!ELEMENT add - - (path, widget-def?)>
<!ELEMENT tag-text - O EMPTY>
<!ELEMENT attribute - - (#PCDATA)>
<!ELEMENT component-path - - RCDATA>
```

Figure 4.1: DTD for Interface Interpreters

the syntax for *tkmap*, a language whose programs interpret for SGML documents (the syntax is very loose, but providing a stricter syntax would have significantly expanded the size of the DTD).

Interpreters of type *tkmap* take a document and convert it to a set of widgets using the Tk[43] toolkit. This is a weakness of our current implementation, but we are examining the STFP part of the DSSSL proposal as a means of specifying a more independent geometry. Lines 1 to 4 enumerate the widget classes used. (These are fairly generic; the real Tk dependence comes in the attributes (lines 13 - 15) which define strings to be passed to the widget constructors.)

The default VM state contains one object, a top level window (a window which is not contained in any other). During interpretation, widgets can be added to this window, or additional top level windows can be generated and used to hold information. An interpreter architecture restricting all UIs to a single top level window would exclude `toplevel` from the content model of the `tkwidget` element. Adding more widget classes expands the possible interfaces.

Before interpreting the document parse tree, an interpreter can add widgets to the initial environment, such as UI elements generic to an application class (eg., menu bars and certain buttons), or which will be used by other widgets constructed during the parse (eg., a list to contain a table of contents for the document). These widgets are enumerated in a component list (`comp-list`). Since Tk widgets all form a tree of strings, each widget requires a name as well as a definition. The name for a default widget must be in the program text, but the name of a widget constructed during the parse of the document may also come from the element's character data or from an attribute.

The definition of a widget has three parts:

1. The *widget class*. This element has three optional attributes which contain Tk specific strings for the creation of the widget.

2. A list of subcomponents describing the widget's children. These subcomponents are defined in the same way as the parent.

3. Any event bindings, such as mouse clicks, particular letters, etc. As Tk only runs on X Windows, these bindings are specific to that environment. The bindings are given names accessible to the application for the callback. As these names are not specific to X, the bindings do not imply an application dependency on X.

Once the initial set of widgets is created, the document parse tree is traversed, starting at the root. At each point there is a particular *tag map* in force which indicates how each tag is to be handled. These maps nest, so the handling of a particular tag can be different depending on which subtree it is encountered in. There are three parts to the handling of a particular tag:

1. A (possibly empty) set of *before* commands. These commands are executed when the node containing the tag is first encountered. In particular, these commands can create new widgets that will be needed for traversing the subtree routed at that node. The available commands will be described below. Any new widgets created at this level are accessible while traversing the subtree as the *parent set*. This parent set is the child of the parent set that was in force before the commands were executed. The initial environment becomes the parent set of the root node of the document, so it is possible to traverse up the ladder to the starting environment.

2. An optional tag map to be used while traversing the subtree rooted at that element.

3. A (possibly empty) set of *after* commands. These commands are executed after the subtree has been traversed and can be used to organize the results of the subtree traversal, similar to the way a parent widget might alter the sizes of its children. These commands have access to any new widgets created by the children

The part of an interpreter actually affecting the current environment is contained in the *commands* associated with the tags. *Tkmap* supports a very small set of commands, reflecting its role as a system for enumerating a tree, not for performing sophisticated computation. Each command takes two arguments:

1. A *path* expression, which leads through the current environment to a particular widget. The path either starts with the root of the enviroment and works its way down the tree, or starts with the parent set and works its way up some number of levels before starting down some part of the widget hierarchy. A path is therefore a series of zero or more `up` elements followed by a list of strings and integers separated by periods. Each `up` ascends one level through the tree of parent sets. Each element in the list descends one level, either choosing the child of the current widget with the same name as the current element in the list, or the nth child. The end of the path is a particular location in the widget hierarchy.

2. A *value*, where that value is either text or the definition of a subtree of widgets. If the value is text, then it either the character data of the element, or it is the value of one of its attributes. If the value is a widget subtree, then the widgets will be created and placed in the hierarchy as specified by the command.

```
<!ENTITY % entry-type "string | number | date | boolean">
<!ELEMENT form O O (section)+>
<!ELEMENT section - O (heading, instruction?,
                        (%entry-type;)*)>
<!ATTLIST section name ID #REQUIRED>
<!ELEMENT heading - - RCDATA>
<!ELEMENT instructions - - RCDATA>
<!ELEMENT entries O O (%entry-type)*>
<!ELEMENT (%entry-type;) - O (label, instruction?, value?)>
<!ATTLIST (%entry-type;) name ID #REQUIRED>
<!ELEMENT entry-list - O (%entry-type;)*>
<!ELEMENT label - - RCDATA>
<!ELEMENT instruction - - RCDATA>
<!ELEMENT value - - RCDATA>
```

Figure 4.2: Sample DTD for Entry Forms

The three available commands are *set*, *append*, and *prepend*. *Set* places its *value* argument at the location in the widget hierarchy designated by the *path*, *append* adds its *value* to the end of the list of children of the *path*, and *prepend* places its *value* at the head of that list.

4.1.6 Generating an Interface

The general interpreter architecture described here is very powerful, although the language defined by the tkmap DTD is not. Nevertheless, it can already be used to describe a large variety of UIs which do not require general computational ability. This section provides an example “logical interface” and shows three similar interpreters creating three different concrete UIs for the same application.

Figure 4.2 provides a small DTD for creating entry forms. A *form* has several `sections`, each of which has a `heading`, an optional `instruction`, and some number of `entries`, where an entry is one of `string`, `number`, `date`, or `boolean`. Each entry has a `label` and may (optionally) have an `instruction` field and and a default `value`. A small medical form with three sections and eight entry fields

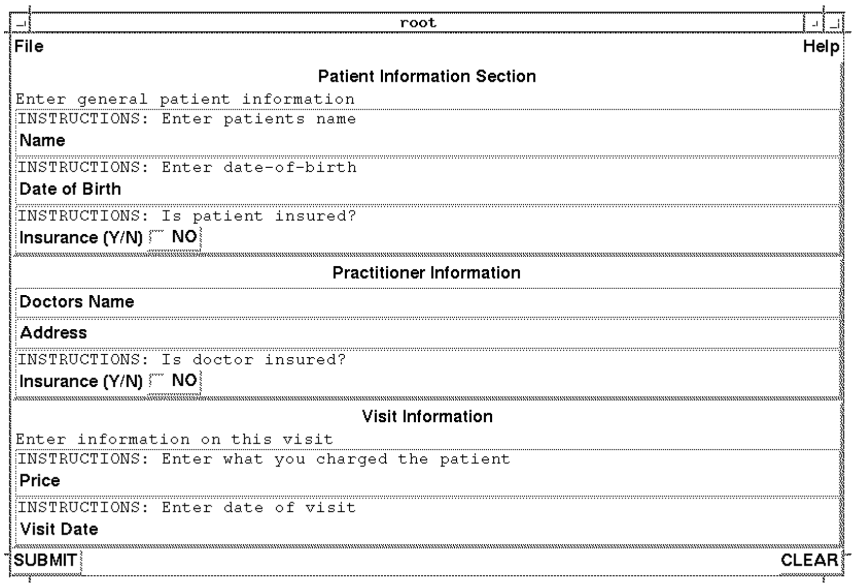
```
<!doctype form SYSTEM>
<section name = patient-info>
    <heading>Patient Information Section</>
    <instruction>Enter general patient information</>
  <string name = pat-name><label>Name</>
        <instruction>Enter patients name</>
  <date name = dob><label>Date of Birth</>
        <instruction>Enter date-of-birth</>
  <boolean name = insured><label>Insurance (Y/N)</>
        <instruction>Toggle if patient is insured, else no.</>
<section name = practitioner>
    <heading>Practitioner Information</>
  <string name = doc><label>Doctors Name</>
  <string name = address><label>Address</>
  <boolean name = insure><label>Insurance (Y/N)</>
      <instruction>Toggle if doctor is insured, else no.</>
<section name = visit-information>
    <heading>Visit Information</>
    <instruction>Enter information on this visit</>
  <number name = price><label>Price</>
      <instruction>Enter what you charged the patient</>
  <date name = dov><label>Visit Date</><instruction>Enter date of visit</>
```

Figure 4.3: Sample Document

conforming to this DTD is presented in figure 4.3.

A straightforward interpreter, which formats all this information in a single window, is contained in figure 4.5, and a generated UI in figure 4.4. `SHORTREF`s (not included in the DTD given here due to length) have been used liberally to give the appearance of a program and considerably lessen the number of keystrokes. After the `doctype` declaration is the name of the document type being interpreted, in this case, `form`. Next comes the list of default widgets. All widget subtrees are surrounded by square brackets. Finally comes the tag map. Tag maps are surrounded by curly braces. Each tag name is preceded by "`--`" and followed by a colon.

The default widget set consists of a frame (`topframe`) containing a menu bar (with two pulldown menus), a frame into which the sections can be placed, and


root

File Help

Patient Information Section

Enter general patient information

INSTRUCTIONS: Enter patients name

Name

INSTRUCTIONS: Enter date-of-birth

Date of Birth

INSTRUCTIONS: Is patient insured?

Insurance (Y/N) NO

Practitioner Information

Doctors Name

Address

INSTRUCTIONS: Is doctor insured?

Insurance (Y/N) NO

Visit Information

Enter information on this visit

INSTRUCTIONS: Enter what you charged the patient

Price

INSTRUCTIONS: Enter date of visit

Visit Date

SUBMIT CLEAR

Figure 4.4: Basic UI

another frame containing `submit` and `clear` buttons. The name of a widget comes from either the program text, the character data of the current widget, or one of the widget's attributes. If it comes from the text, then it is the quoted string immediately preceding the widget type. If it is the text, then the tag `tag-text` is used, and if an attribute, the tag `attribute` followed by the attribute name and a colon.

The outermost tag map gives commands for the different entry types (only `string` is given in the figure, but the others are identical, except `boolean`, which is a checkbutton), `labels`, `values`, `instructions` and `sections`. Each entry tag creates a new frame in the current section and creates `label` and `value` widgets for it. Encountering the `label` or `value` tags fills in the text for the appropriate widgets. Locating an `instruction` tag within an entry prepends a text field containing the instruction to the frame.

When a section tag is encountered, it adds a new frame to contain its information. A new tag map containing definitions for `heading` and `instruction` is

```
<!doctype tkmap SYSTEM>
form:
<widget-list>
["topframe"<frame>
 ["menubar"<menubar packstr = "-fill x">
  ["file"<menubutton packstr = "-side left" textstr = File>
   "help"<menubutton packstr = "-side right" textstr =Help>]
  "sections"<frame>
  "buttons"<frame>
    ["submit"<button packstr = "-side left">
     "clear"<button packstr = "-side right">]]]
{--string:<append><path>^1,
     [<attribute>NAME:
       <frame initstr = "-relief ridge -borderwidth 2" packstr = "-fill x">
          ["LABEL"<label packstr = "-side left">
           "VALUE"<entry packstr = "-side left">]];
  ...

 --value:<set><path>^1.VALUE,<tag-text>;
 --label:<set><path>^1.LABEL,<tag-text>;
 --instruction:
     <prepend><path>^1,
        ["INSTRUCTION"<text initstr = "-height 1 -width 30"
                            packstr = "-fill x" textstr = "INSTRUCTIONS:">];
     <append><path>^1.INSTRUCTION,<tag-text>;
 --section:
     <append><path>TOPFR.SECTIONS,
        [<attribute>NAME:
             <frame initstr = "-relief raised -borderwidth 3"
                      packstr = "-fill x">];
     {--heading:<append><path>^1,["HEADING"<label>];
                <set><path>^1.HEADING,<tag-text>;
      --instruction:
           <append><path>^1,
              ["INSTRUCTION"<text initstr = "-height 1">];
           <append><path>^1.INSTRUCTION,<tag-text>;}
}
```

Figure 4.5: Interpreter for Basic UI

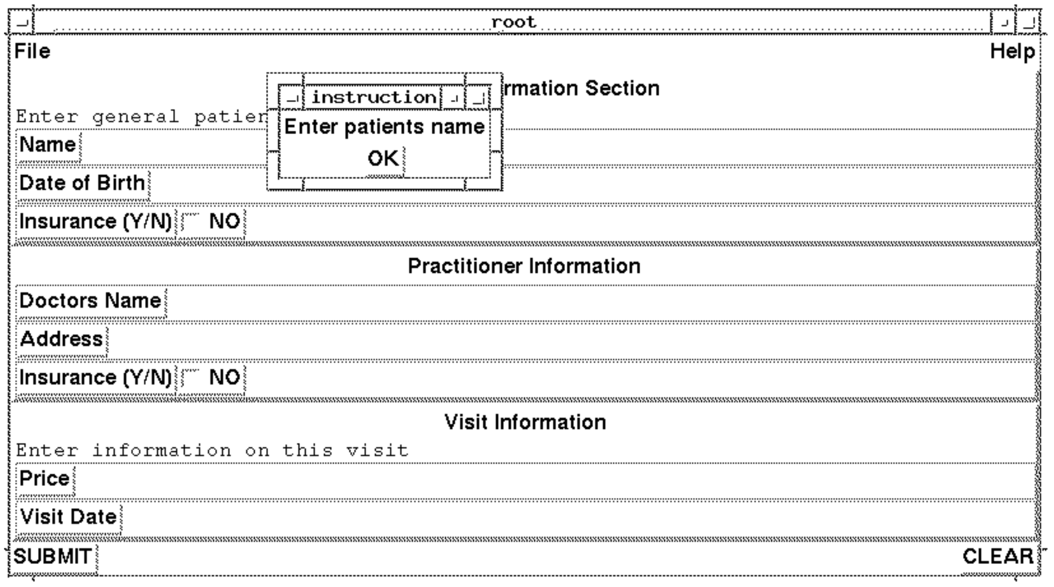


Figure 4.6: UI with Hypertext

introduced. Since tag maps are lexically scoped, `instruction` appears in the interface in two separate ways: instructions associated with an entry are prefaced with "INSTRUCTIONS:", since they refer to the outermost tag map, and instructions at the beginning of a section are not, since they refer to the inner tag map.

Figure 4.7 contains a fragment of an interpreter that creates the variant UI in figure 4.6. The only changes are to the commands associated with the `entry` and `instruction` tags; `label` becomes a widget of class button instead of class label, and instructions are placed in separate pop-up windows, providing a more hypertext style interface. The differences between the interpreters are minimal; the application doesn't even need to be aware of the difference if the widget set can internalize the differences.

Another alternative is provided by the fragment in figure 4.9, resulting in figure 4.8. Each section now has its own top level window, instead of being contained in `topframe`. A new button, `Sections` is now placed in the menubar. This button brings up a menu which serves as a table of contents for the sections; each section

```
{--string:<append><path>^1,
     [<attribute>NAME:
       <frame initstr = "-relief ridge -borderwidth 2" packstr = "-fill x">
          ["LABEL"<button packstr = "-side left">
           "VALUE"<entry packstr = "-side left">]];

...

 --instruction:
      <append><path>^1,["INSTRUCTION"<toplevel>
                          ["INSTR"<label> "OK"<button>]];
      <set><path>^1.INSTRUCTION.INSTR,<tag-text>;

...

}
```

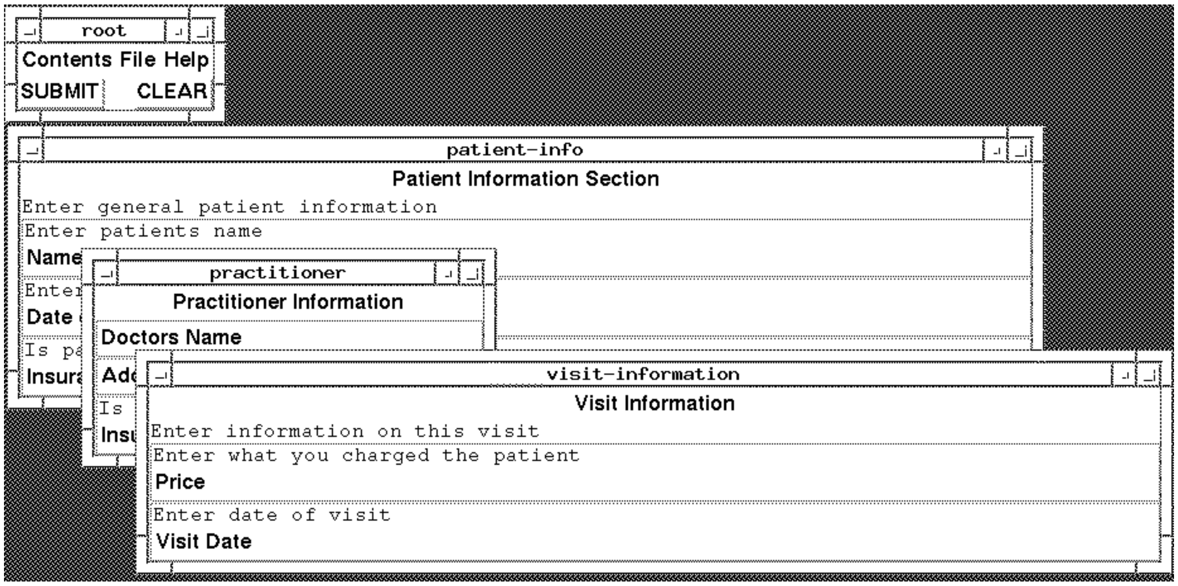
Figure 4.7: Interpreter for Hypertext UI

adds a button to the menu which pops up the section's window. Each entry now appends itself to the `entries` frame in its section's window.

Although these alternative grammars represent significant changes to the look and feel of the interface, they do not represent a change to the logical interface requirements underlying all three; the application is not required to be aware of differences so long as any additional events created by the look and feel are handled outside of the applications purview.

4.1.7 Interaction between an Application and its Interface

Applications interact with their UIs at three particular points: when setting callbacks, when processing a callback, and when modifying the UI. This section discusses these in turn, with an emphasis on graphical applications.



Patient Information Section

Enter general patient information

Enter patients name

Name

Practitioner Information

Doctors Name

Visit Information

Enter information on this visit

Enter what you charged the patient

Price

Enter date of visit

Visit Date

Figure 4.8: UI with Multiple Windows

Setting Callbacks

After the entire interface is interpreted, the root widget is handed back to the application. The concurrency control layer is now able to set callbacks through interrogating the widget hierarchy. The application expects to find a particular tree of widgets and events corresponding to its logical requirements. The tree returned, however, may be signficantly larger than this, if a local interpreter was used instead of the default one. So long as the desired tree is unambiguously embedded in the actual tree this is not a problem. The application can descend the hierarchy and place callbacks at the desired nodes.

Receiving Callbacks

The signature of our callback functions is standard and should be platform independent. So long as callbacks do not rely on specific structures, such as the X event structure, there should be no difficulty in migrating to all platforms. Current experience, though, is limited to Unix-based X Windows toolkits. The SGML-related

```
form:
["topframe"<frame>
 ["menubar"<menubar packstr = "-fill x">
  ["file"<menubutton packstr = "-side left" menu = file
          textstr = File>
   "sections"<menubutton packstr = "-side left" menu = sections
              textstr = Sections>
       ["sections"<menu>]
   "help"<menubutton packstr = "-side right" menu = help textstr =Help>]
   "buttons"<frame>
    ["submit"<button packstr = "-side left">
     "clear"<button packstr = "-side right">]]]

...

{--string:<append><path>^1.ENTRIES,
...
 --section:
      <append><path>.,
          [<attribute>NAME:
               <toplevel initstr = "-relief raised -borderwidth 3"
                      packstr = "-fill x">
                 ["entries"<frame>;
                  "ok"<button>;]]
      <append><path>TOPFRAME.MENUBAR.SECTIONS.SECTIONS,
          [<attribute>NAME:<button>];
      <set><path>TOPFRAME.MENUBAR.SECTIONS.SECTIONS.0,<attribute>NAME;
...
}
```

Figure 4.9: Interpreter for Multi-Window UI

techniques discussed here, however, are independent of X and should generalize easily.

Modifying the Interface

Once created, the UI remains an active entity in the environment. GUI widgets maintain links both to the application objects they represent and to the original document locations from which they were generated. Certain widgets also retain the virtual machine environment and tag map that existed when they were created. The links from the interface back to the application objects (which don't need a GUI to communicate among themselves) allow the user to direct cooperation among objects. Other information, such as found in a document, has no way of communicating other than through its own structure, which can only be provided by refering back to the original (or its parse tree).

This information supports the operations of context-sensitive *cut and paste* and *drag and drop*, essential to creating a completely integrated environment. Without making these operations context sensitive, only isolated fragments of information, such as text strings, can be passed around among applications. When they are context sensitive, then the operations can be seen as manipulating coherent chunks of information, such as groups of objects, or portions of documents. At the source, it is important to know just what information is in the scope of the operation; at the destination we need to know what type of information is being received to determine how to integrate it with the local environment.

Drop and *paste* are variations on the creation of the UI. In both cases, a source object appears (whether an OO-type object or a document) which needs to be integrated into the existing environment at the destination point. We can distinguish between an object-to-object communication, which doesn't require any alteration to the GUI, and one in which new information needs to be displayed at the destination. In the first case, the system only needs to locate the application object represented by the destination widget and invoke the appropriate method. This is done by firing the event for the *drop* or *paste* method of the widget, passing in the source object as a parameter. If the communication does not result in any change in appearance for the user it does not interest us further here. In the second case, the extra information left from the original parsing is necessary, and demonstrates one of the advantages of our approach.

When some SGML information is to be integrated at some location in the application, it should meld semantically at that location as if it had been at that location in the original document, and not added later. To do that, we need access to the VM state available at that location and to the tag map in force at that location. Without the VM state, we don't know what objects to integrate the new information with, and without the tag map, we don't know *how* to integrate them. The tag map provides the semantics for the interpretation. These two elements almost form a *continuation* for the interpreter at the point at which the destination was created (an actual continuation would consist of the VM state, the tag map that would be in force after a subtree is interpreted, and the next node to be processed after the subtree is completed). Container widgets maintain this information. When information arrives, it is in the form of a document subtree. The current GUI subtree is removed, and the document subtree is interpreted and integrated at that location.

Removing a subtree (*cut*) may also be a complex operation, as a document subtree may be reflected in may parts of the UI. In the third interface from our example, a menu contains a table of contents for the sections; removing a section also requires modifying that menu. Complicated situations may require a special interpreter for removing a subtree. In the general case there is no guarantee that integrating a document in a GUI is reversible.

4.1.8 Expanding the interpreter

The examples used here have been kept simple for purposes of exposition, but more sophisticated cases are straightforward extensions of the same principles. We have been working on several extensions which not only support generating a greater variety of interfaces, but which also support active documents by generally treating documents as applications:

- Add local variables to the interpreters. Any truly complex functionality requires temporary storage of local computations, implying the need to define variables. Variables global to an interpreter are defined before any widgets are defined. Other variables are local to the processing of a particular tag and are declared before any of the commands for that tag. Variables follow the usual lexical scoping rules.

- Add more powerful primitives. To support general computation, interpreters need *if-then-else* and looping constructs, as well a larger set of types, such as numbers, strings, lists and vectors. One promising approach is to "compile" interpreters into Dreme, which would let interpreters support Dreme's (and therefore Scheme's) syntax and semantics. An important extension would be to include the data types and primitives defined in the DSSSL standard.

- Expand the *path* element to a more general tree search mechanism. In the interpreter, paths are used to navigate around the output tree. If we expand the interpreter to include variables, etc., there may be any number of trees to navigate, including the input document. We plan to significantly strengthen this feature.

- Make interpreters callable from each other. The structure of an interpreter is sufficiently regular that it is straightforward to build a calling mechanism among interpreters. The caller needs to supply a pointer to a subtree in the document and an initial VM state. Other parameters may be passed as well. Submaps can also be placed in local procedures. Combining this feature and the previous would allow the same piece of subtree to be evaluated several times, particularly interesting if the source text is a program containing a loop.

Notably lacking from this list are expanding the number of object types for the interpreter and the operations they support, i.e., the VM. These really depend on the domain of a particular class of intepreters. We anticipate that different application areas will have different domains; interpreters in these domains will manipulate the corresponding virtual machines, sending transactions over in the form of documents. A document may also be processed in more than one domain. The same document can be displayed, stored, forwarded, and used to trigger other activities. Since documents and interpreters do not have any direct access to the virtual machines, except as provided by the local host, the local system can exercise complete control over the behavior of a document.

4.1.9 Expansion to multiple users

Everything discussed so far has been in the context of a single user, although Dreme is intended to be distributed. Extension to multiple nodes is very straightforward: an object can send an interface description to any UIMS of which it has the address. For example, a button press by user A can cause a window to pop-up on the screen of user B by having the callback pass a UI description to B's UIMS. The callbacks can all be sent to B when they are set, as they would naturally migrate.

Alternatively, if it has permission, A's system could migrate the entire code containing both the UI description and all the callbacks to B so that it would all be present when the interface appears.

4.2 The Ultimate Web Browser

In 1989, Tim Berners-Lee (?) created the World Wide Web (Web), a distributed hypermedia system consisting of, in particular, a transport protocol, *http*, and a markup language, HTML, based on SGML. HTML documents are displayed in a Web *browser* and contain hyperlinks to other documents on the Internet. When the user activates a link, a request goes to an *http daemon* on the other side of the link, which retrieves and returns the requested document. The *browser* then displays the retrieved document. In other words, the Web implements a distributed shared address space for HTML documents and, like Dreme, uses mobility to distribute computation around the network. We could trivially combine the two by supporting a superset of HTML and considering an HTML document as a particular kind of inactive Dreme object. The previous sections, however, lay the groundwork for a much more powerful environment which allows us to move beyond both HTML and application interfaces to display and manipulate a much wider range of information.

Since 1989, Web traffic has grown to be a significant part of Internet traffic. As the complexity of Web documents has grown and attempts to use the Web protocol for other applications have increased as well, the limitations of HTML have become apparent. As a result the Web community has proposed a new standard, HTML+, to compensate for these deficiencies. Nevertheless, HTML+ still suffers from the main failing of HTML – in order to be displayed by the Web, data must be translated into HTML(+), leading to a loss of information regarding the data's structure. An application designed to manipulate data of a particular type would need to reparse the data. In fact, it might require a sophisticated expert system to retrieve the original structure; there may be any number of mappings from a particular format to HTML, and there is no way to tell which of those mappings a particular data supplier used (HTML itself contains no hint of what that might be).

The spread of HTML is probably having an unnecessary chilling effect on the development of application specific data formats, particularly in SGML, because a tramslator to HTML is needed before documents are accessible through the Web; the obvious increase in the quantity of available information is hiding a decrease in quality. Since public domain SGML parsers are available, the technical problem seems to be that separating the presentation semantics from the document requires requires executing the presentation code. Of course, current Web browsers (except tkWWW) are not interpreters, but Dreme processes are, and we have shown how they can be extended as necessary.

By placing the approach of the previous section in the context of the WWW, we arrive at the intelligent Web browser, capable of receiving information in a variety of formats and manipulating that information as desired, whether that is simply to display, or to place some of a document in a database, send some in email, and place other pieces in various places in the user's environment.

By default, three elements must be necessary to display any document:

1. the document
2. the document type description
3. a default interpreter for displaying documents of the given type.

If these three are present, then the document can be considered to have a minmal presence on the Web; it can be browsed. The other manipulations mentioned represent desirable additional functionality, but are not essential to maintaining upward compatibility with the current Web. We must now show how these basic pieces can be mapped into the Web, and then how the additional functionality can be supported.

The universal mechanism for addressing in the Web is the Universal Resource Locator, or URL; either our extensions can be mapped into URLs, or they cannot be supported by the Web. URL syntax, fortunately, is very flexible. From the perspective of a browser, an URL consists of a string with, at a minimum, the protocol for accessing the object (such as *http* for files accessed through standard Web servers, or *ftp* for those accessed through that protocol), the Internet node holding the file, and the local path and file name. The file name extension indicates the data type of the file (eg., `file.html` or `file.ps`).

We can now prescribe the addition of four new suffixes:

1. `.sgm` indicating that a document is SGML compliant. Proper HTML documents also fit into this class, but their handling is already optimized by existing browsers, and there is no reason to lose that ability. This is intended for the large class of documents which will utilize "application specific" DTDs.

2. `.dtd` indicates a file containing a DTD. The rest of the filename must contain the name of the document type. Servers now provide various means for redirecting URLs, so the DTD need not be actually present.

3. .dis indicates a display interpreter. The filename must include the name of the document type. The filename would ideally include the widget set being used for the interpreter; our current implementation does not do so.

4. `.int` indicates that the file is an interpreter. The filenameis not required to indicate the document type, as that is part of the interpreter text.

This section has described how to add general SGML-compliant text to the WWW. Adding application-specific interpreters written in Dreme is straightforward on the browser side. Sharing these interpreters through the Web, whether freely or for financial gain, will be more complex. The interpreters are also not general distributed applications; although they may arrive over the Web, and may access URLs, all the computation occurs in one location. At some point the more general power of Dreme will be necessary.

4.3 Combining Menus with Events

User actions add an element of concurrency by creating multiple simultaneous dialogs with an application. So long as these dialogs can be circumscribed within certain limits (such as being described by regular expressions) where all the possible actions are known beforehand, the concurrency can be simulated by the interface. However, if the complexity grows beyond that, then real concurrency must be supported.

This problem is compounded by Dreme's inherently multithreaded architecture. Any number of objects may be simultaneously present, each with its own interface, and each of these might need to support a multithreaded dialog. We will exploit the simplicity of the menu approach for each individual thread within the context of an overall event loop through the judicious use of continuations for callbacks. In essence, before the system processes the next event, all the state that was available to the old-style menu driven application is stored in a continuation. When the desired event is received by the system, it can be passed to the continuation so that the event can be handled in the appropriate context. A side effect of this approach is the ability for application code to interact synchronously with the interface, i.e., a callback can be the continuation of a function call, making it appear like any other I/O operation.

Another form of interface concurrency is created by the interaction between applications. For example, a user might perform some operation, such as *drag and drop*, exposing some piece of interface to two objects. Now, one event can cause changes in both applications, which can be reflected back to the interfaces of the other applications.

This concurrency can not be handled generally by any *à priori* analysis of the application, as the various objects to be displayed cannot be known ahead of time. Two standard approaches to handling this problem are:

1. Have only one application that knows all the types of objects that might be displayed, so it can keep track of them with its own data structures. This severely limits the kinds of objects that can be displayed, but still allows them to communicate with each other, as they are all in the same application. However it necessitates extra application constructs to handle the concurrency.

2. Create independent processes to handle each concurrent object or dialog. In Unix this would be accomplished by using the `fork` command. This handles the concurrency by shifting the burden onto the operating system. It also allows a more heterogenous set of objects to be displayed. However it supplies no support for communication among objects except in a hierarchical fashion (i.e., each object passes return information to its parent), implies considerable overhead due to context switching, and does not easily support any shared context among all the processes.

Each of these approaches has its drawbacks with respect to our goals. Dreme already allows objects from various sources to coexist and communicate. We will extend this to the interface; all the objects of all applications share the same space and therefore communicate in a straightforward fashion. But to maintain this desirable feature, we must first understand the kinds of actions that will arise in the interface and then determine the constructs which can handle them within a single Dreme process.

4.4 A continuation-based user-interface

As has been noted before, continuations are considered difficult and perplexing to program with. It's scarcely reasonable to hold out the promise of new flexibility only to offset it with even more new complexity. We resolve this by presenting the user with an abstraction of how events are related to each other and an API that implements this abstraction. Manipulating continuations is pushed beneath the API and out of the programmer's purview.

4.4.1 A taxonomy of events

In order to handle concurrency in the interface, we must first understand how user-generated events can lead to concurrency. We consider only those (user) events that lead to actions by the application, as only they lead to changes of state. In this section we will give a short taxonomy of these actions to motivate the description of how to handle their concurrency. We define an event as some external stimulus to the application, such as a mouse click, and an action as a piece of application code, identified by name, which can be called in response to an event.

An action has a *lifespan* during which it can occur; it is *born* at some point, exist for some period of time (which might be forever) and then *die*. During its lifespan, an action is, at various points, either *active* or *inactive*. This is the action's *state*. When an interface object comes into being, its actions are born as well, and when it is destroyed, its actions die as well.

For the purpose of analyzing concurrency, we can further categorize live actions by where they stand along four axes:

1. *global/local*: This characterizes the *scope* of an action. A *global* action is always alive once it comes into being, ordinarily at the beginning of an application. It remains continually active unless explicitly deactivated. A *local* action arises at some point in a computation due either to an event (eg., a popup menu), or to the flow of the application, and dies at some later point. A *local* action's state can be changed either explicitly or implicitly.

2. *synchronous/asynchronous*: A *synchronous* action returns to its caller after it executes. An *asynchronous* action does not. Some part of the interface, such as a popup, may also be synchronous; it can be called by an application and will return a value, just as any regular function does. The asynchronous counterpart of this returns immediately, although it may create other interface components that continue to exist independently. Asynchronous actions are similar to callbacks in ordinary event-loop systems.

3. *unique/repeatable*: A *unique* action is essentially non-reentrant. Once a unique action has started, and until it completes, user events cannot trigger that action again. An example might be a worksheet to calculate the value of an entry in a tax form application. Since the entry can only have one value, it would not make sense calculate the same value twice in parallel. A repeatable action is one which can have any number of simultaneous instantiations – the action can be invoked any number of times before any of

the instantiations complete, such as the invoices in our example.

4. *blocking/nonblocking*: A *blocking* action is one which inhibits a group of related actions. An example would be recalculation in a spreadsheet; while the recalculation takes place, values cannot be altered. Actions which do not block are simply *nonblocking*. Certain blocking actions are also *exclusive*, meaning there must be no outstanding actions for them to be active.

Not all actions stand at the corners of this hypercube. An action might arise with other actions and outlive their demise. It may be possible to activate repeatable actions only a limited number of times, either fixed in the application or dependent on other factors. Blocking actions might only block certain other actions, and not all.

In this system, old-style menus have local, unique, blocking actions. No such obvious model is available for graphical interfaces; their actions are a mixture. As we have seen, though, it is only with great difficulty that they can be made synchronous. An application may have some pull-down menus that are available throughout the run of the application: they are global. Popups are local and frequently synchronous. Control panels are usually global, unique (there is only one for the application), and nonblocking. On the other hand, in many applications there are dialog boxes where the entire application is blocked until the user responds.

We will now apply these concepts to simplify the programming of the application. The techniques presented here make it possible to program it in a style similar to an ordinary program, with the concurrency usually implicit rather than explicit. The fundamental concept of using continuations for callbacks, however, is extremely powerful. Continuations allow a program to jump around in fairly arbitrary ways. Very different abstractions from the ones presented here could be developed under the same overarching framework.

4.4.2 An overview of the toolkit

In describing the toolkit we will first examine how the events are built, organized and connected to GUI components, after which we will show how the code in the example actually looks and explain how callbacks are created. The toolkit makes extensive use of continuations. As has been noted before, continuations are considered difficult and perplexing to program with. It's scarcely reasonable to hold out the promise of new flexibility only to offset it with even more new complexity. We resolve this by presenting the user with an abstraction of how events are related to each other and an API that implements this abstraction. Manipulating continuations is mostly pushed on the other side of the API and out of the programmer's purview.

Our toolkit has been implemented on top of both the Athena and Tk widget sets. This is possible because the toolkit functions as a scheduler for actions, determining whether they are available or not, with little concern for what triggers the action, what occurs during the action, or what the underlying GUI looks like. To convey how the connection to a particular widget set is accomplished, we will describe the connection to Athena. Support for Motif would be almost indistinguishable.

The Dreme/Toolkit Interface

The Dreme interpreter provides direct support for C++ classes, so the widgets are first wrapped in C++ and then in Dreme. The description that follows will attempt to be as X independent and as Dreme independent as possible and can be easily generalized, although we will start with an brief explanation of the Xt interface.

Athena widgets are created with default callbacks for most events. The special behavior of a particular application is largely accomplished by substituting a different function to be called in place of the default one. Since a Dreme function cannot be so easily substituted for a C language one, the interface must be a little more complicated.

All callbacks to Dreme code are handled by a pair of C++ functions:

1. the `uniCBfunc` function, which receives all callbacks that are to be handled by Dreme functions. The `clientData` parameter, which is specified by the user when creating an Xt callback, contains a pointer to a pair whose `car` contains the Dreme callback function and whose `cdr` contains its user-specified argument. The body of the function sets two global Dreme identifiers, `xtcbfunc` (for the callback function) and `xtcbarg` (for the single argument), to these two values, respectively.

2. the `addCallBack` method, which takes as arguments a Dreme function and a Dreme object. These are `cons`ed and become the client data of the Xt callback. The method finally calls `XtAddCallback` for the appropriate widget.

As is common in Unix, the X Window server communicates with clients, such as a Dreme process, through a socket. Since this is also the way that Dreme processes communicate with each other, Dreme simply adds this socket to the list that the *read-eval-print loop* waits for and assigns it a special thread, called the `xtloop`. When input is waiting on this socket, `xtcbfunc` is set to `#f` and the application context object is told to process one event. After the event has been processed, `xtcbfunc` is examined. If it is not `#f`, then this is a Dreme callback and `(apply xtcbfunc xtcbarg)` is called to handle it. Otherwise the system waits for another event.

The `xtloop` always returns to the Dreme level after allowing Athena to perform whatever handling the toolkit requires for a particular event, rather than having the Athena callback recursively invoke the Dreme interpreter. As has been mentioned before, Dreme (and GUI's) are multithreaded, and Dreme uses continuations extensively. If the interpreter were called recursively, managing the process stack, as well as messages from other Processes and continuations, would become extremely complex.

This event loop is quite straightforward and is handled as a thread by the Dreme scheduler, although it is usually blocked for input. The next section will show how it can be made multi-threaded.

4.5 A continuation-based user-interface

As has been noted before, continuations are considered difficult and perplexing to program with. It's scarcely reasonable to hold out the promise of new flexibility only to offset it with even more new complexity. We resolve this by presenting the user with an abstraction of how events are related to each other and an API that implements this abstraction. Manipulating continuations is pushed beneath the API and out of the programmer's purview.

4.5.1 A taxonomy of events

In order to handle concurrency in the interface, we must first understand how user-generated events can lead to concurrency. We consider only those (user) events that lead to actions by the application, as only they lead to changes of state. In this section we will give a short taxonomy of these actions to motivate the description of how to handle their concurrency. We define an event as some external stimulus to the application, such as a mouse click, and an action as a piece of application code, identified by name, which can be called in response to an event.

An action has a *lifespan* during which it can occur; it is *born* at some point, exist for some period of time (which might be forever) and then *die*. During its lifespan, an action is, at various points, either *active* or *inactive*. This is the action's *state*. When an interface object comes into being, its actions are born as well, and when it is destroyed, its actions die as well.

For the purpose of analyzing concurrency, we can further categorize live actions by where they stand along four axes:

1. *global/local*: This characterizes the *scope* of an action. A *global* action is always alive once it comes into being, ordinarily at the beginning of an application. It remains continually active unless explicitly deactivated. A *local* action arises at some point in a computation due either to an event (eg., a popup menu), or to the flow of the application, and dies at some later point. A *local* action's state can be changed either explicitly or implicitly.

2. *synchronous/asynchronous*: A *synchronous* action returns to its caller after it executes. An *asynchronous* action does not. Some part of the interface, such as a popup, may also be synchronous; it can be called by an application and will return a value, just as any regular function does. The asynchronous counterpart of this returns immediately, although it may create other interface components that continue to exist independently. Asynchronous actions are similar to callbacks in ordinary event-loop systems.

3. *unique/repeatable*: A *unique* action is essentially non-reentrant. Once a unique action has started, and until it completes, user events cannot trigger that action again. An example might be a worksheet to calculate the value of an entry in a tax form application. Since the entry can only have one value, it would not make sense calculate the same value twice in paral-

lel. A repeatable action is one which can have any number of simultaneous instantiations – the action can be invoked any number of times before any of the instantiations complete, such as the invoices in our example.

4. *blocking/nonblocking*: A *blocking* action is one which inhibits a group of related actions. An example would be recalculation in a spreadsheet; while the recalculation takes place, values cannot be altered. Actions which do not block are simply *nonblocking.* Certain blocking actions are also *exclusive*, meaning there must be no outstanding actions for them to be active.

Not all actions stand at the corners of this hypercube. An action might arise with other actions and outlive their demise. It may be possible to activate repeatable actions only a limited number of times, either fixed in the application or dependent on other factors. Blocking actions might only block certain other actions, and not all.

In this system, old-style menus have local, unique, blocking actions. No such obvious model is available for graphical interfaces; their actions are a mixture. As we have seen, though, it is only with great difficulty that they can be made synchronous. An application may have some pull-down menus that are available throughout the run of the application: they are global. Popups are local and frequently synchronous. Control panels are usually global, unique (there is only one for the application), and nonblocking. On the other hand, in many applications there are dialog boxes where the entire application is blocked until the user responds.

We will now apply these concepts to simplify the programming of the application. The techniques presented here make it possible to program it in a style similar to an ordinary program, with the concurrency usually implicit rather than explicit. The fundamental concept of using continuations for callbacks, however, is extremely powerful. Continuations allow a program to jump around in fairly arbitrary ways. Very different abstractions from the ones presented here could be developed under the same overarching framework.

4.5.2 An overview of the toolkit

In describing the toolkit we will first examine how the actions are built, organized and connected to GUI components, after which we will apply this to our original example.

Our toolkit was originally built on top of the Athena widget set. As our Scheme interpreter provides direct support for C++ classes, widgets were first wrapped in C++ and then in Scheme. More recently we have been migrating to Tk, but this does not affect the current discussion. The description that follows will attempt to be as X independent as possible and can be easily generalized.

Defining actions

A toolkit, such as Athena, usually has a set of low level widgets which can be put together by an application to create more coarse grained objects. It is these that are significant to the application; our efforts are concerned with aiding in manipulating these larger objects. Furthermore, the elements of an application's UI are not designed as independent entities, but often have some kind of nested structure; for example, a event in one window can trigger an action that affects the appearance of another. We find nested scopes work well for supporting a hierarchy of windows, whether in a single application or spread around a network. The actions in an application are grouped in recursive lists that function as scopes, which we call *lexical coordination structures*. A single window may contain events triggering actions in one or more scopes, with the parent scope in a parent window. A user event in a child window can trigger an application action in a parent window by name. Scheme, our implentation language, also has nested scopes (as well as first class continuations), but there are reasons to have a parallel hierarchy of scopes in the user interface:

1. There may not be an appropriate one-to-one mapping from the organization of the application to the appearance of the user-interface.

2. There may be significant interface management associated with an action, such as displaying or removing windows. This is exactly the kind of management from which user interfaces typically try to relieve the programmer.

3. In a distributed application, the user interface may be the best, if not the only, way for parts of the application to communicate. This, in fact, is evident in the X Windows inter-client communication mechanism.

To maintain independence from a particular widget set, a GUI is composed of three layers:

1. A widget layer, whose job is to appear on the screen and pass events up to the next layer. When an event occurs, a particular scope is called in the upper layer with the name of the event.

2. A scheduling layer, composed of nested scopes of actions, where each action has been defined with the appropriate set of attributes. The innermost scope receives a call from the widget layer that an event has occurred – this is the only callback allowed at the widget layer – and looks up the action associated with that event. According to the state of the system and the attributes of the action, it is determined if the action is active. If so, the necessary changes to the application state are made (essentially setting flags) and the callback

of the action is invoked, either synchronously or asynchronously, as the case may be. Finally, any necessary cleanup operations are performed, and the result returned to the caller, if appropriate. There should be a one-to-one, or one-to-many correspondance between this layer and the first – events at the GUI level which don't have a corresponding action are not handled by the application. We concentrate on this layer here.

3. An application layer, which has access to both of the lower layers. From a coordination standpoint, the interesting communication is between the application and the middle layer, but the application also needs to be able to affect the GUI to provide feedback to the user. How dependent the application is on a particular GUI platform depends on the nature of this communication.

In this framework, a graphical application starts with the application setting callbacks in the coordination layer. Before a GUI component is displayed, the application layer passes it to the coordination layer, which sets all the GUI callbacks back into the coordination layer and then displays it. When an event occurs with an associated action, the application is called with the name of the action, any client data for the action, a pointer to the current scope in the application, a pointer to the widget associated with the action, and an optional, system dependent, event structure (if the action is called directly by an application component, then the last two parameters are not included). If this widget is associated with the creation of new widgets (such as a pop-up window), then these the new scope for the new widget and the new widget itself are passed in.

A lexical coordination structure is created by a call to `make-event-scope` passing in an optional parent scope and a list of action descriptions. An action description has several attributes to determine the behavior of the action. These are listed

Attribute	*Default*	*Values*	*Description*
name	none		Name of the action
active	yes	yes, no	Whether action is active at startup
unique	multiple	single, multiple	If action can be called a second time before the first finishes executing
meet	asynch	synch, asynch	If action is synchronous or asynchronous
blocking	no	yes, no, excl	Whether this action blocks other actions when active
terminal	no	yes, no	Whether action terminates the associated interface component.
enabling	no	yes, no	Introducing a set of callbacks active until the action occurs.

Figure 4.10: Action Attributes

in figure 4.10, along with their default values. The description need only list attributes different from the defaults. A *terminal* action removes the associated set of windows from the screen. If the container widget is part of a synchronous action, then when the terminal action occurs, the result of the associated action will be returned as the result of the call which created the container. Any action description may itself contain, as an optional parameter, a list of action descriptions. In that case, the action introduces a new scope (frequently associated with a popup window).

Setting callbacks

We will now apply this scheme to the original bridge problem. We will start with the standard, single-threaded bridge game described above, and then relax the constraints until almost everything occurs simultaneously. Our implementation language is Dreme, but SML/NJ, a Standard ML compiler with first-class continuations, or its derivative, Concurrent ML, are logical alternatives, as could be prototype-based object oriented languages, such as Self (Chambers, 1989). Using a language-independent structure has the potential advantage of allowing application components in several different languages to cooperate in a distributed system.

```
(define play-bridge
  (lambda (...)
    (letrec ((the-scope (scope-maker ´((rubber #f 0 (unique single))
                                        (tutor #f 1 (meet asynch)))))
             (rubber (lambda ...))
             (tutor (lambda ...)))
      ((the-scope ´find-name ´rubber) ´set-callback! rubber)
      ((the-scope ´find-name ´tutor) ´set-callback! tutor)
      ...
      (the-scope ´display))))
```

Figure 4.11: Playing Bridge

Another important aspect of the toolkit is that actions can be triggered either by a user event or by invocation by some other part of the application.

In the basic bridge example, we mentioned two actions at the outermost level, both of which are *active*:

1. Start a rubber. This action is *unique* and *asynchronous*. In other words, it can be chosen immediately after the application starts up, but there can only be one rubber at a time, and it return value, the final score, is discarded when it terminates.

2. Start a tutorial. This action is *multiple* and *asynchronous*, as we've allowed for concurrent sessions with the tutor.

The code for this in figure 4.11. The scope is a list of action descriptions. An action description has the following optional fields:

1. The name of the action.

2. A path through the GUI to the interface component which corresponds to the event that invokes the action.

3. A default value for the action. This is the value returned by the action if has not been assigned a callback function.

4. A list of attributes. This need only contain those attributes with values other than the defaults.

5. An optional scope, if the action introduces an inner scope.

A false value in the second field indicates an action which is not represented directly in the GUI. If only the first two fields have values, then the description refers to an event in an outer scope, but there is some user event at this level which can initiate it. This will be the case here; button for both `rubber` and `tutor` will appear on user's screens, but the action are held by the bridge game object. Each action has a separate function as callback.

Having specified these initial settings, we can alter their relationships without altering their implementation, to a certain degree. In a tournament, `play-bridge` would return a score to the tournament manager. This is accomplished by changing the definition of the scope to:

```
´((rubber #f 1 (unique single)(meet synch)(terminal yes))
  (start-tutorial #f 1 (meet asynch))
```

In this case, at the conclusion of a rubber, any interface components associated with the rubber are removed and the resulting score will be returned to the toura-ment. The The net result of this change is that the display call to `the-scope` will return synchronously with the result of `start-rubber`. The tournament calls `play-bridge` synchronously. The actual play of a rubber is initiated by one of the players.

```
<!entity % suit "club | spade | heart | diamond">

<!element bridge O O (url*, addtl*, (bid, hand, board)?, hand?)>
<!element hand - O (%suit;)*>
<!attlist hand name ID #IMPLIED>
<!element (%suit;) - O EMPTY>
<!attlist (%suit;) name CDATA #IMPLIED
                   val  CDATA  #IMPLIED>

<!element bid - - ((%suit;)*, value*, choices*)>
<!element value - O EMPTY>
<!attlist value name ID #REQUIRED
                val CDATA #REQUIRED>
<!element choices - O EMPTY>
<!attlist choices name ID #required>

<!element board - O (curr-bid?,player+,hand?)>
<!element curr-bid - O EMPTY>
<!element player - O EMPTY>
<!attlist player position (north | south | east | west) north
                 bidder (bid-yes | bid-no) bid-no
                 dummy (dummy-yes | dummy-no) dummy-no>

<!element my-hand - - (hand)>
<!element url - - rcdata>
<!attlist url url CDATA #required
              name ID #required>
<!element addtl - O (#PCDATA)>
<!attlist addtl name ID #required>
```

Figure 4.12: Bridge DTD

It might also be considered unfair to use the tutor while playing a rubber. To accommodate this, starting a game and starting a tutorial are made *blocking*. In the current implementation, calling help would then need to be placed in an outer scope, so that it is not blocked as well. Concisely fine tuning these dependencies will be an important area for extending this model. Nevertheless, we can achieve large changes in behavior with a small effect on the implementation.

To play bridge, each participant must retrieve the bridge interface and (eventually) his hand. The bridge DTD and interface interpreters are in figures 4.12 and

```
bridge:
<widget-list>
["buttons"<frame>
 "help"<toplevel startup = no>
 "sections"<frame>
 ["label1"<label initstr = "Current Bid">
  "bid"<toplevel startup = no>
  "ready"<toplevel startup = no>
   ["press"<button textstr = "Ready To Start A Hand?">]
   "play"<toplevel startup = no>
   ["press"<button textstr = "Play a Card">]
  "currentbid"<entry>]]
{--url:<append><path>buttons,
     [<attribute>NAME:
       <button  packstr = "-side left">]#
 --addtl:<append><path>buttons,
     [<attribute>NAME:
       <button  packstr = "-side left">]#
 --pcdata:<set><path>^1,<ttext>#
 --hand:<append><path>^sections,
     ["hand"
       <frame>
        ["spades"<frame initstr = "-relief ridge -borderwidth 1">
           ["label"<label initstr = "Spades">
            "cards"<frame>]
        ...]]#
      {...
       --diamond:<append><path>^hand.diamonds.cards,
              [<attribute>NAME:<button packstr = "-side left">]#}
```

Figure 4.13: Bridge Interface Interpreter (con't)

```
 --bid:<append><path>sections.bid,
        ["ding"<frame>
        ["info"<label textstr = "Your Turn To Bid">
         "ready"<button>
         "bidpop"<toplevel startup = no>
          ["instr"<label textstr = "Select your bid:">
           "suit"<frame>
              ["name"<label textstr = "Pick a suit:">
                ...
               "club"<button packstr = "-side left">]
            "numtr"<label textstr = "Select number of tricks:">
            "tricks"<frame>
            "othsel"<label textstr = "Other selections:">
            "choices"<frame>
            "OK"<button packstr = "-side left">]]]#
       {--value:<append><path>^ding.bidpop.tricks,
                 [<attribute>NAME:<button packstr = "-side left">]#
        --choices:<append><path>^ding.bidpop.choices,
                 [<attribute>NAME:<button packstr = "-side left">]#}
 --board:<append><path>sections,
        ["sections"<frame>
         ["players"<frame>]]#
 --player:<append><path>^sections.players,
        [<attribute>POSITION:<frame>]#
        ...
 --diamond:<append><path>sections.hand.diamonds.cards,
        [<attribute>NAME:<label>]#}
```

Figure 4.14: Bridge Interface Interpreter

```
<!doctype bridge SYSTEM>
<url name = rubber
    url="http://bridge.com/nort.sgm">Start a Rubber</>
<url name = help
    url="http://bridge.com/help.sgm">On-line Help</>
<addtl name = tutor>Tutor
```

Figure 4.15: Initial Bridge Interface

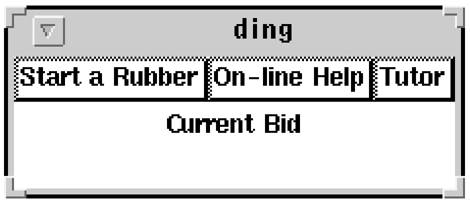


Figure 4.16: Bridge Startup Screen

4.13. A bridge interface arrives in three separate pieces during a rubber:

1. The outer interface, defined in figure 4.15, only includes the help pages and buttons for starting the rubber and calling the tutor. for invoking the tutor. This is seen in figure 4.18

2. The player's hand and the bidding interface come in a second document. This document is integrated with the existing interface.

3. The dummy's hand.

We will concentrate on explaining the bidding.

The local scope is shown in figure 4.20. The bidding event, which does not appear initially, contains all the events for bidding. The bidding event is actually called from a method invoked by the game object. Pressing the OK button will return the choice to the game.

The code for bidding is shown in figure 4.21. Whenever a bid is needed, the thread enters a do loop (line 4) until a valid bid is returned. Each time through, the

```
<!doctype bridge SYSTEM>
<bid>
<value name = "one" val = 1>
<value name = "two" val = 1>
<value name = "three" val = 1>
<value name = "four" val = 1>
<value name = "five" val = 1>
<value name = "six" val = 1>
<value name = "seven" val = 1>
<choices name = "pass"></>
<hand>
<spade name = ONE val = 1>
<spade name = THREE val = 3>
<spade name = FIVE val = 5>
<heart name = ONE val = 1>
<diamond name = TWO val = 2>
<diamond name = FOUR val = 4>
<diamond name = FIVE val = 5>
<diamond name = ACE val = 13>
<club name = ONE val = 1>
<club name = THREE val = 3>
<club name = FIVE val = 5>
<club name = QUEEN val = 11>
<club name = KING val = 12></>
<board><player north><player east>
<player west><player south>
</board>
```

Figure 4.17: Bridge Hand

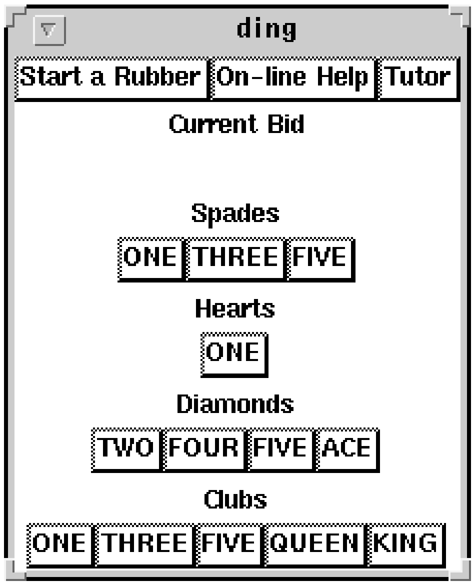


Figure 4.18: Bridge Startup Screen

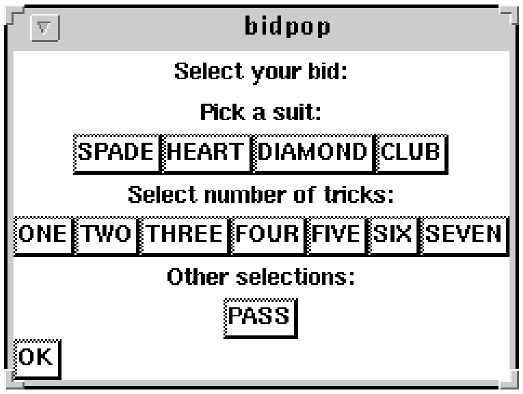


Figure 4.19: Getting a Bid

```
´((rubber "BUTTONS.RUBBER")
  (tutor "BUTTONS.TUTOR")
  (help "BUTTONS.HELP" 1
     (unique single) (meet synch))
  (bidding "SECTIONS.BID" 2
     (unique single)(meet synch)
     (scope "SECTIONS.BID"
         (ready "DING.READY" 4
             (unique single)(meet synch)(blocking yes)
             (terminal yes)
             (scope "DING.BIDPOP"
                 (spade "SUIT.SPADE" 0)
                 (heart "SUIT.HEART" 1)
                 (diamond "SUIT.DIAMOND" 2)
                 (club "SUIT.CLUB" 3)
                 (one "TRICKS.ONE" 4)
                 (two "TRICKS.TWO" 5)
                 (three "TRICKS.THREE" 6)
                 (four "TRICKS.FOUR" 7)
                 (five "TRICKS.FIVE" 8)
                 (six "TRICKS.SIX" 9)
                 (seven "TRICKS.SEVEN" 10)
                 (pass "CHOICES.PASS" 11)
                 (ok "OK" 12
                     (unique single)(meet synch)
                     (terminal yes)))))))
```

Figure 4.20: Local GUI Scopes

```
((my-scope ´find-name ´bidding)
 ´set-callback!
 (lambda (obj scope newgui)
   (do ((suit #f suit)
        (tricks #f tricks)
        (choice #f choice))
       ((or (and (and suit tricks) (not choice))
            (and choice (not (or suit tricks))))
        (begin
          (if choice choice (cons suit tricks))))
     ((scope ´find-name ´ready)
      ´set-callback!
      (lambda (obj scope newgui)
        (map (lambda (cb val)
               ((scope ´find-name cb) ´set-callback!
                      (lambda x (set! suit val)
                              (display (list ´set-suit val)))))
             ´(spade heart diamond club) ´(0 1 2 3 4))
        (map (lambda (cb val)
               ((scope ´find-name cb) ´set-callback!
                      (lambda x (set! tricks val)
                              (display (list ´set-trick val)))))
             ´(one two three four five six seven) ´(1 2 3 4 5 6 7))
        ((scope ´find-name ´pass) ´set-callback!
                  (lambda x (set! choice ´pass)))
        (scope ´display newgui)))
     (scope ´display newgui))))
```

Figure 4.21: Getting a Bid

bidding event sets a callback for the ready action, which corresponds to a button on the popup informing the user that it is his turn to bid. When that button is pushed, callbacks updating the values of `suit` (line 14), `tricks` (line 19), and `choice`(line 24) are created for the main bidding popup. They function asynchronously until OK is pressed, bringing down all the windows associated with bidding and terminating an iteration of the do loop. If the loop variables have correct values, the bidding callback returns that to the game. Other elements of a synchronous bridge game proceed similarly, however this does only serialized the main thread of the bridge game. Help and tutoring can proceed simultaneously if desired.

As mentioned, each rubber is composed a number of games, each of which returns a score. Traditionally, each game is synchronous and unique. This opens two possible implementations:

1. Place all the logic directly in the function body. This is the approach taken in the psuedo code of the first chapter.

2. Create nested scopes for the actions that take place within a rubber. In this case, actions can be controlled either programmatically or through the user interface.

In the former case, little changes behaviorally, but in the latter, if the game action is made multiple, then any number of games can be active simultaneously. Updating the score would need to be placed in a monitor to avoid contention if two games end almost simultaneously. The same effect can be obtained by placing the update into another, unique, action. Other changes would need to be made to the source code, but the basic control flow could be maintained. The approach allows an application to migrate back and forth between serial and parallel implementations.

The major change from parallelizing rubbers is the need to rethink the rules of the game to accomodate this new possibility. We leave this question to the interested parties.

Within each game, actions are also normally synchronous. The same technique for parallelizing each rubber can be used for parallelizing each game and trick, so all tricks of all games can be played at simultaneously. Starting a trick can be done by either a user or the application. Within that, the order of play can also be parallelized. Supporting such extreme parallelism requires extra work to determine termination. We are looking at extending the model to support some of this; for example, there are exactly 13 tricks, and each trick uses four cards, and each card comes from a different user. It should be possible to automate all three of these constraints.

The one aspect of a bridge game which *is* inalterably unique is the playing of a card – each card can appear in at most one round of play in a single game. In our system, we model this as a unique action which never terminates. For each user, for each game, there are thirteen unique actions, corresponding to the cards. They are accessed through a synchronous action, corresponding to a popup. The continuation associated with playing a card terminates the synchronous action without returning, so the card-playing action never terminates until garbage collected after the game.

4.5.3 Combining continuation code with existing systems

It is an important side-effect of this approach that we can combine it with existing applications that were written using an event-loop. With the event-loop, each action functions as a discrete unit and only communicates with other actions through shared data structures. If we add new actions which do not write to any of those structures, such as we have been discussing, then the existing actions are unaffected.

Similarly, we can take any set of actions and rewrite the callbacks along the lines we have laid out without affecting other actions, so long as we update the shared data structures correctly. It is also possible to proceed in the reverse direction by taking the Scheme code for some set of actions, rewriting it in CPS and then converting that to C++.

We have taken advantage of this to build the GUI over the Athena and Tk widget sets. Callbacks are handled by the widgets themselves, except where they are overridden by Scheme callbacks. The same approach can be used with any other application, so long as there is a way for Scheme code to access and manipulate data in the language of the application.

The most complicated part of the interface, in fact, is the relationship between Scheme's read-eval-print loop and the GUI toolkit's event-loop. They cannot mutually call each other forever, so one must submit to the other and be called as a routine. As both Athena and Tk give access to the event-loop, we have chosen to replace the event-loops with event-loops in Scheme that function as threads. When an event occurs, it is first passed to the GUI. If there is a Scheme callback, then two Scheme variables are set, which are used to execute the callback when the toolkit returns. This eliminates concern the for process stack, and has the added feature of allowing both toolkits to coexist simultaneously.

It is also possible to give priority to the event-loop. By using continuations as callbacks, we have already placed the entire state of the computation in an object. The Scheme inner loop needs to be modified to call this continuation. In addition, the Scheme inner loop needs to know when to return to the event-loop. Inside the toolkit, whenever the application needs input, is a call to `get-events`. This call must terminate the Scheme inner-loop, but the rest of the Scheme code need know nothing of this.

4.6 Related Work

Our work has dealt with two separate aspects of user interfaces which we must consider:

1. The visual appearance of a graphical user interface.

2. Control flow of the application from user events.

We will treat each in turn.

4.6.1 The Appearance of a User Interface

In order to satisfy the requirements of supporting applications distributed across heterogenous platforms, we have attempted to completely avoid the question of visual appearance and, instead, focus on transmitting the logical interface, independent of any particular GUI platform or toolkit. This is in accord with the original philosophy of SGML, but appears to be unusual in GUI developments. Logical requirements for an interface are certainly a part of GUI design, and style guidelines [18] may approach this, but there does not seem to be any GUI work in "executable specifications." It is more frequently argued that designing a user interface is an iterative, experimental process that would be difficult to automate [40, 53].

Providing a textual GUI description language, separate from application behavior, is not unusual. Motif, the OSF standard interface for X Windows, includes the User Interface Language (UIL). UIL provides a small language with keywords for all the Motif widgets, bindings, and other pertinent information which developers can use to specify window appearance, and some interaction, in separate text files. These files are run through a separate "compiler" and are then opened by the application at run time. Another system for X windows is the Widge Command Library(?) (WCL) which extends the X Resource Database to include the entire widget hierarchy. Version 4 of the Actor language provided a Window Description Language, similar to UIL, for MS Windows. To a certain extent, HTML can be seen as providing the same function for hypertext. Because of the general utility of a textual representation, there are probably also innumerable undocumented examples in the private sector.

Each of these systems is designed to provide an interface for a particular application in a particular GUI toolkit. A UIL description, for example, is not easily transported beyond Motif. This drawback can be partly overcome by porting the toolkit to every interesting platform, as is proposed for Fresco, a CORBA compliant GUI toolkit for X11R6[37, 38]. There are also some commercial products that have taken the same approach. However, these are currently all compile-time libraries.

All of these approaches are still tied to the display of graphical widgets and are too low level for our purposes, which are really concerned with the semantics of the interface. In addition, these systems, with the partial exception of HTML, are tied to single applications. If there is an explosion of applications on the network, it will not be possible to slowly handcraft user interfaces one at a time; the SGML based approach we advocate allows interfaces for an entire range of applications to be specified simultaneously, as well as support functionality which cannot be derived at the widget level. Nevertheless, the ready availability of a multi-platform toolkit would simplify matters.

4.6.2 The Behavior of a User Interface

Because of its complexity, extensive work has been done on scripting user interface behavior. Until recently, this work has been mired in dealing with a single threaded environment, as we have alluded to previously. We will discuss some of the earlier work, and then describe more recent multithreaded work.

The Model-View-Controller (MVC) GUI architecture developed for the Smalltalk-80 system does not address threading, but is otherwise somewhat similar to our approach. MVC attempts to provide a clean break between the information to be displayed and its semantics, on the one hand, and the appearance of that information on the screen, on the other. The former element is called the Model, the appearance is the View, and the Controller is a mapping from user actions to functions in the Model. The normal flow of control is that a user event triggers the Controller to send a message to the Model, which in turn modifies the View. This paradigm allows several different Views of the same model to exist simultaneously on the screen, each View providing a different display semantics. Views and Controllers normally are developed in pairs, as the user's actions will depend on how information is displayed. In our system, the Model is provided by the application and/or the SGML document passed to the local interpreter. The View, the underlying widgets, is generated by the interpreter. The interpreter, however, also creates the Controller simultaneously, although the application is also capable of changing the Controller by changing the states of events. Although this is a logical path for development, and the MVC controller paradigm has been very successful in the Smalltalk community, other GUI paradigms are also possible.

The Network Extensible Window System[27](NeWS), from Sun Microsystems, is probably the system most similar to our efforts in many areas. Had NeWS been more successful, our approach to the User Interface might have been quite different. NeWS assumed the existence of a network separating the application from the display, so that communication would involve interprocess communication, as it does for X Windows. The first interesting element of NeWS is the adoption of PostScript, the page description language from Adobe, as its display language. Because of preexisting work making PostScript device independent, this made NeWS more easily ported to a variety of system. It also meant that an application GUI was a piece of code sent from the application to the display server.

NeWS offers significant extensions to PostScript to manage GUI interactivity. There are two that are fundamental:

1. The server runs any number of lightweight PostScript threads. By adding a *fork* command and monitors, an application's GUI running at the server is completely multithreaded. This multithreading does not necessarily spread to the client.

2. These threads can receive events. Postscript code normally writes to a printer and does not receive input. In NeWS, PostScript threads can declare their interest in different events and will be informed when those events occur. As any thread can express its interest in any event, this implies that different clients can react to the same event, as a form of implicit IPC.

The NeWS approach to GUI programming saw the GUI Server and its client(s) as communicating parallel processes. They are connected by a two way pipe. The client can send a byte-stream of PostScript commands to the server, and the PostScript routines can, in turn, send another stream back in the other direction, just as they could write to any other port. This does not enforce a structured communication, but lets each client establish its own protocol. NeWS attempts to support a more structured format by also supporting tagged data packets moving from the server to the client. The client can then instruct the server code about what tags to place in communications and then use these to determine what is being communicated.

NeWS was a very ambitious system, certainly more ambitious than X11, which has become the standard Unix windowing environment. If one were to consider the NeWS server as capable of functionality beyond just display, and there is no particular reason why it couldn't be used for arbitrary computation, then there is no fundamental difference in capabilities between NeWS on the one hand and any system using remote execution, such as Telescript, Obliq, or Dreme, on the other. Obliq and Dreme provide very different models for the way this functionality is achieved, but it is not known what the difference between the Telescript and NeWS models are, although Telescript wishes to be seen as the Postscript of the networked world.

From the perspective of GUI systems, our approach is similar to NeWS in the transmission of programs over the network and support for multithreading. The fundamental difference between the two is Dreme's lack of explicit support for the concept, inherited from Andrew, of the networked window server. In this model, also found in X11, the window server runs on the user's machine and applications all run in separate processes scattered around the network. The ability to reduce network traffic by placing more of the processing burden in the server is a basic justification for the NeWS architectre. However, to place any significant application logic in the server required writing it in PostScript, a separate language from the rest of the application. This distinction between server and application is not particularly meaningful Dreme, since elements of the application can migrate around the network as necessary without needing to change language. We have also developed a more sophisticated model of controlling user events in a multithreaded environment.

Another multithreaded graphical user interface residing on similar principals to our approach, although not explicitly, is *eXene*[51], developed in Concurrent ML(CML). This is an extension of Pegasus[39], written in PML, a predecessor of CML. Before describing eXene and Pegasus, I'll briefly describe CML.

CML is an extension of Standard ML and is implemented on top the SML/NJ compiler, a continuation-passing style compiler which inspired the Dreme interpreter architecture. Because of its CPS architecture, SML/NJ could easily add continuations to the SML language. CML takes advantage of them to provide threads in the same fashion as Dreme (and innumerable other Scheme systems); where CML differs from other systems is in its extension of the ML typing system to cover concurrency, particularly communications between threads. All communications in CML are through synchronous channels, as in CSP. If one thread of a communication arrives before the other, the first blocks waiting for the second. CML introduces the *event*, a new first-class object representing a potential communication. Examples of events are `receive` and `transmit`, where the former represents receiving a value over a particular channel and the latter represents sending a value over a particular channel. For these events to actually occur, the program must accomplish two more tasks. First, `wrap` is used to bind a particular event to a function, eg. `wrap (receive inCh, fn x => x)` represents passing the value coming from `inCh` to the identity function. Second, this pair must be passed to a synchronizing construct, such as `sync` or `select`, which actually causes the thread to block waiting for the (an) event. `Sync` waits for a particular event, and `select` chooses one of a list. CML events are unusual because they are first-class, and can therefore be passed around. CML provides several other constructs with variations on this functionality.

Dreme can support the functionality of CML first-class events, although without static type checking, by providing closures to act as the desired objects. Channels are essentially closures which must be called by both the sender and receiver with entry controlled by a `monitor` (so they don't miss each other) and the continuation of the first caller held. Of course, this also a reflection of the fact that Dreme and CML (within the scope of a single process) are based on very similar constructs.

In Pegasus, each window starts three threads, one each for keyboard events, mouse events, and control events, as well as any additional ones needed for coordination and maintaining state. Widgets do not use callbacks to communicate with their clients, but instead synchronize across typed channels. This handles many of the GUI programming problems that have been mentioned repeatedly. Some multithreaded applications, such as an interactive multi-view graph editor, have been developed on Pegasus. `eXene` extends Pegasus by providing a multithreaded X display replacing the C-language Xlib.

The Pegasus/eXene work targets lower level graphics functionality, and cannot interact with existing systems, although it does provide the added safety of static type checking. Dreme, on the other hand, is able to coexist with existing GUI toolkits, but without type safety.

More interesting is comparing Pegasus event handling with Dreme's. Pegasus depends on CML events (actually PML, which is a subset). As noted above, these can be implemented in Dreme, so the same mechanism could be used in both cases. In the Pegasus approach, however, the coordination among events must be handled by the code. This is not a problem if the coordination is simple. But when the relationships become complex, this can become a burden, requiring the programmer to build automata, even with the advantage of multithreading. Creating a separate layer with a specific notation automates this. If the software is mostly composed of reused elements, then the coordination layer may represent most of the development effort. It shoud be entirely possible to create a separate coordination layer for Pegasus in CML.

Chapter 5

Distributed Garbage Collection

5.1 Introduction

In a language such as Dreme, where objects and object references are scattered around a large, network of autonomously evolving elements, there is no escape from garbage collection. For Dreme to be a viable element of the distributed systems of the future, it must be possible to perform efficient distributed garbage collection, including collection of cycles, with low network overhead.

We present a decentralized distributed garbage collection (DGC) scheme for distributed mobile objects. This algorithm is unique in that it collects *all* garbage, whether cyclical or not, without requiring (as do other algorithms) the coordination overhead of a finding a distributed root set (a set of objects around the network known to be active from which garbage collection starts).

It is our contention that in dealing with networks of this size, bottom-up algorithms that do not depend on any *a priori* global data structures, or other long-term coordination will prove to be more robust. The algorithm in this paper requires little coordination beyond the already existing links between the nodes involved, and extends only as long as the algorithm requires.

The algorithm can be seen as an extension of Piquer[44] and Birrell et al.[5], both of which augment reference counting (RC) algorithms by maintaining the distributed inverse reference graph (IRG): each object, α, maintains a list of pointers to other sites known to have references to it. In the former case references can be replicated across sites, forming a tree, while the latter ensures that α always knows every site containing a reference to it. Neither algorithm is able to collect cycles. We will show there is enough information in the IRG to collect cycles with a reasonable overhead.

Maintaining the IRG has several ancillary benefits not normally considered:

1. The same information can be used to maintain consistent states among replicated objects, as in a software *cache consistency* protocol [11].

2. Nodes can implement a variety of garbage collection strategies, such as charging for the continued existence of an object, migrating objects to accessor nodes, or arbitrarily collecting an object and informing its referents of their dangling pointers. These choices are not necessarily reasonable within a single applications but may be important among distributed objects communicating across security domains.

3. Objects know which nodes are sending DGC information and can choose to accept or discount that information accordingly. This provides the possibility of discerning and responding to malicious behavior.

The remaining sections of this chapter will first give a summary of other proposed DGC algorithms, then briefly outline the basic concept for our algorithm and describe the complete algorithm for a single collection and prove its correctness. It will then expand the algorithm to handle concurrent DGC operations and explain how to avoid deadlock. The final section will show some heuristics to determine when to start a collection operation.

5.2 Related Work

Distributed garbage collection algorithms generally follow one of the two techniques used for collecting a single address space: mark-and-sweep or reference counting. Mark-and-sweep algorithms [33, 34, 45] first choose an area of distributed memory (although not blocking processing), determine the distributed *root set*, and organize inter-node references. Then they determine the transitive closure of the references from the root set. Everything else is garbage. Reference counting techniques [15, 13, 44, 54, 5] require much less coordination. However, they do not locate distributed cycles. Both kinds of algorithms generally have two phases. In the local phase, completely unreferenced objects are garbage collected. In the second phase inter-node information is propagated through the network, identifying more garbage.

Ladin and Liskov[33] gives an algorithm for collecting cyclical garbage using a client/server model. Distributed references are tracked by the server, which can use them to determine where garbage is. For scaling to larger networks, the article proposes dividing the network into *areas*, each with its own server and a global server to sit above for inter-area references. Puaut[45, 46] provides another client/server model for distributed collection of active objects. This extends Kafura's [29] algorithm for collecting agents by allowing lost messages and removing synchronization constraints. The various nodes of a system send timestamped information to a global collector containing intra-node edges. As the objects are active, the head and tail of each edge is a pair giving the object's name and its state (one of *root, inactive, running,* or *unknown*). Unlike ordinary objects, an active object is not necessarily garbage if not reachable by a persistent root; an active object may send its address to another object at some point, making it reachable from a persistent root. A collector for active objects must handle this case. Both of these methods require dependence on a third party (the global collector), and on that party's judgments concerning the behavior of objects. Their scaleability is open to question, as they require cooperation from the entire network.

Lang et al.[34] extends mark-and-sweep to collecting cyclical garbage in large networks. Processes are organized in hierarchy of ever larger groups, with the largest containing the whole network. A coordinated mark-and-sweep is performed at each level to eliminate garbage from the group, until finally the largest group removes all garbage. The use of hierarchical groups has several drawbacks:

1. Responsibility is diffuse; nodes are dependent on decisions of unidentified nodes.

2. The entire network must agree to submit to same the algorithm. In a system as open as the Internet, it is unreasonable to assume that a single algorithm for a particular functionality will be optimal in all situations. A decentralized algorithm which depends only on point-to-point communication among the "interested parties" promises to be easier to integrate with nodes implementing special purpose algorithms.

3. Garbage may need to be examined several times before being definitively established as garbage.

Traditional RC requires *increment* and *decrement* messages; incorrectly ordered messages can lead to collection of live objects. Distributed RC algorithms, starting with weighted reference counting (WRC) [4, 59], generally eliminate the *increment* messages. In WRC, each object is given a large value which is spread around when a reference is copied. Each outstanding reference contains a weight, and all these weights add up to the weight of the object. Decrement messages are sent when remote references are collected and are accompanied by the former reference's weight. Because weights cannot be divided when they sink to one, the initial algorithm required indirection cells, proxies for the object with their own trees of references. A number of improvements and alternatives [15, 13, 44] have eliminated these. None of these algorithms handles cycles, and the object must trust the sender of the decrement message.

A different approach starts with Piquer[44] and continues with Shapiro et al.[54]. Scion-Stub Pointer Chains [54] exploit the use of parent pointers (called *scions*) to track where distributed references to an object reside. These parent pointers establish the inverse reference chain from an object to its accessors. Because references or objects may pass from one node to another, these pointers will form a tree rooted at the object. As with traditional RC, garbage is collected from the leaves in; cycles are not collected. Shapiro et al.[54] also shows how correct communication can be assured, even in the presence of lost messages. We will assume the presence of the same mechanisms for collecting non-cyclic garbage. Birrell et al.[5] provides a very similar algorithm, but eliminates the chains of references. To assure that site failures are correctly handled, each object knows all nodes that have references to it, effectively reinstating the the *increment* message; premature collection is prevented by forcing the sender of a reference to keep its copy until receipt is verified.

Incremental algorithms which do not halt processing, such as the one we will present, are frequently explained in terms of a tri-color marking scheme[60, 61]. In such schemes, persistent roots are initially colored *grey* and all other objects are *white*. The algorithm then describes how the children of *grey* objects are also turned *grey*. Whenever all of an object's children are *grey*, the object is turned *black*. At the termination of the algorithm, all nodes are either *black* (alive) or *white* (garbage). *White* nodes can be collected. We will use tri-color marking, but we will assign a different meaning to the colors, as explained below.

5.3 The Basic Algorithm

To simplify the discussion, we will temporarily disregard questions of physical implementation in an actual network. For current purposes, each object can be considered to reside on a separate node. We make the following definitions:

1. Objects are named *a, b, c,* .... Each object has a unique identifier (UI). UI's will be subscripted with the name of the object when necessary to avoid ambiguity.

2. Network nodes are named *alpha*, *beta*, *gamma*, ...

3. We call the traditional reference graph the *forward reference graph* (FRG). We will denote FRG edges with dotted lines.

4. The *inverse reference graph* (IRG) is obtained by switching the direction of all the references in the FRG. We assume this is available for all objects. We will denote IRG edges with solid lines.

5. Objects have a state of either *alive* or *dead.* We will mark objects as either *white, grey,* or *black*.

6. An object which is always alive is called a persistent root (PR). PRs will always be shown in *black.*

7. Since objects can arbitrarily try to learn their status, each such operation is called a Garbage Collection Operation (GCO). The object that starts the

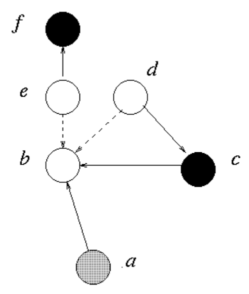


Figure 5.1: Basic Graph

operation is the Garbage Collection Root (GCR). Each GCO may be subscripted by the name of its GCR.

Figure 5.1 shows a reference graph with a number of objects we will use to explain our algorithm. Object *f* is a PR, *a* is a GCR, (*d, b*) is an FRG edge, whereas (*d, c*) and (*c, b*) are IRG edges. Objects *b, d,* and *c* form a cycle. We will reuse the same graph, changing the directions of edges and the persistence of objects.

The fundamental insight behind our algorithm is a dual property of traditional mark-and-sweep. In a traditional mark-and-sweep, the GCR is a PR, the FRG is traversed, all touched nodes are eventually marked *black*, and all *white* nodes are eliminated. Suppose, instead, we choose a GCR which is not also a PR, and traverse the IRG. Since the IRG contains all the objects with references, directly or indirectly, to the GCR, then the GCR is alive if a PR is touched (meaning the PR refers to it in the FRG); otherwise it is garbage. Also, because being alive is transitive, every object on a path from the GCR to the PR in the IRG is also alive.

We will mark objects not involved in a GCO as *white*, objects that are known to be alive as *black*, and other objects in the GCO, whose state is not yet determined, as *grey*. This leads to the following coloring rules for the IRG traversal:

1. At the beginning, all PRs are *black*, the GCR is *grey*, and all other nodes are *white*.

2. If an edge has a *grey* tail and a *white* head, the head turns *grey*.

3. If an edge has a *black* tail and a *white* head, the head turns *grey*. (This rule is technically not necessary, but simplifies demonstrating correctness.)

4. If an edge has a *grey* tail and a *black* head, then the tail turns *black*. In the FRG, this edge has a *black* tail and a *grey* head. Since only PRs are *black* at the outset, these edges are part of the transitive closure of some PR in the FRG.

5. If none of rules 1 - 4 can be applied to any edge, then any remaining *grey* objects can be removed.

These rules are in contrast to traditional tri-color marking algorithms where *grey* objects are always alive. In both cases, at the end of the GCO, no *grey* objects remain.

Figure 5.2 shows four variations of our graph with initial and final colorings.

In the first case, with no cycles, the state of the *grey* nodes is unambiguous; the leaves determine locally that they are garbage and are collected. In doing so they remove links to their children in the FRG, one of whom must be the parent in the IRG traversal. When that IRG parent has heard from all its children (cf. object *b*), it, too, knows itself to be garbage, and communicates that to its parent, etc. At the end of the traversal, no *grey* nodes are left – they've all been collected.

In the second case, there are *grey* subtrees, with one *black* branch connecting the GCR with a PR. As in the previous case, all *grey* objects can be collected during the traversal, so only the live nodes remain at the end.

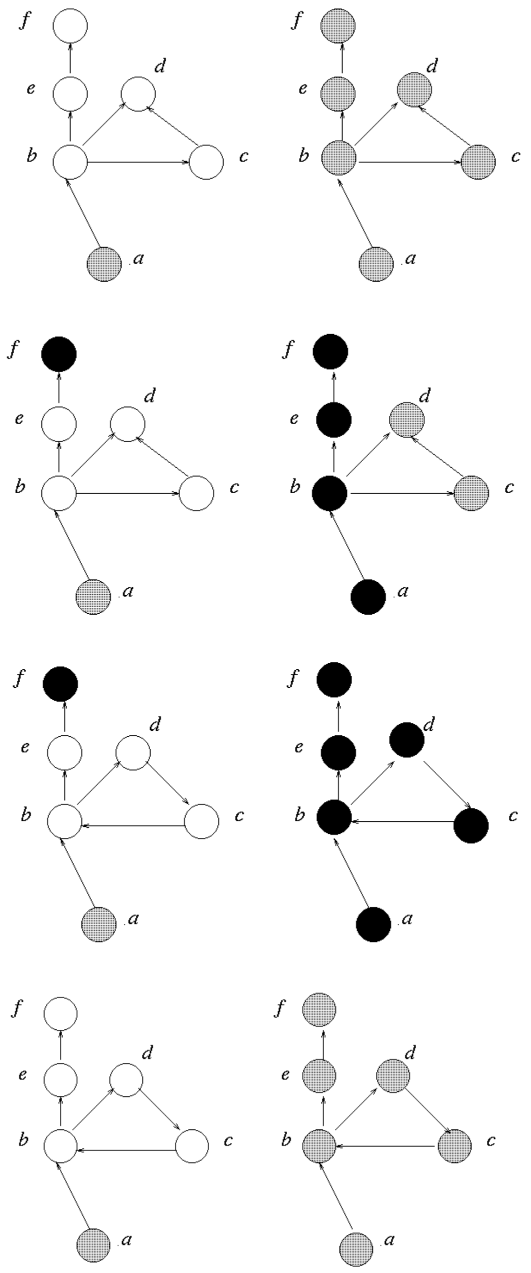


Figure 5.2: Graph Colorings

This lets us add a sixth rule:

- If a *grey* node has no outward edges, it can be collected.

The third case contains a cycle attached to a branch with a PR. Rules 2 and 3 ensure that the entire IRG is marked; if a cycle is encountered, all its members will initially be colored *grey* or *black*. However, if any member is marked *black*, all will eventually be marked *black* by rule 4. In the end all members of the cycle will be correctly marked.

The last case contains a cycle unconnected to a PR. By rule 2, all members are marked *grey*. Since no *black* objects are encountered, and each object has an outward edge, all remain *grey*. Once no more nodes can change color, these are the only *grey* nodes to remain. They can be eliminated as garbage, as in rule 5.

We can summarize the process succinctly with the following steps:

1. An object chooses to start the operation (rule 1).

2. The set of objects, , , which refer to it are located (rules 2 and 3).

3. If any PRs are found to be in , , then we find Δ, the intersection of , with the transitive closure of the PRs in the FRG (rule 4).

4. Any object in , $-\Delta$ is collected (rule 5).

The key to an implementation is ensuring that steps 2 and 3 complete before step 4 is undertaken.

5.4 How to Traverse

There are two possible means of traversal, serial and parallel. As they have slightly different characteristics, we will discuss serial first and parallel afterwards. We will show that applications of rules 2, 3, and 4 are exhausted by the end of the traversal, at which point rule 5 can be invoked. This must be done by the GCR, as only the GCR knows when the traversal has finished.

The complication in a serial IRG traversal is finding the back edges, which indicate the presence of cycles. These occur when a *grey* node tries to traverse a child and finds it is already *grey* (i.e., an ancestor). Each *grey* object must keep track of all the *grey* to *grey* edges for which it is the head, which includes its parent and the back edges from its descendants. In the third example in fingure 5.2, if edge (b,d) is traversed before (b, e), the cycle is traversed and colored *grey* before it is found to be alive. When b turns *black*, rule 4 applies to the edge (c, b). If b does not keep track of c, then it would have to retraverse the subgraph. Object b invokes rule 4 on its back edges. Object c recursively does the same, as do its ancestors in the IRG traversal, back to b. Finally c returns to b, which then returns to its parent, a, which also turns *black*.

During the outward traversal, each object, a, (starting with the GCR) applies rules 2 and 3 to all children, forming a tree from the IRG. By the time the children have returned, rules 2 and 3 cannot be applied in the subtree, as there are no more white nodes. If object a is black, it then applies rule 4 to any back edges. As any back edge comes from an object in the subtree rooted at a, by the time a returns to its parent, no applications of rules 2, 3, and 4 are possible in the subtree rooted at a. As no child returns to its parent until this is true, when the traversal terminates at the GCR, rule 5 can be applied. The algorithm is shown in pseudocode in figures 5.3 and 5.4. Procedure `mark` corresponds to rules 2 - 4, `blacken` colors the back edges, and `terminate` handles rules 5 and 6. The purpose of `GCID` is explained below.

The only restrictions on the order of events is the requirement that all children

```
color myColor := white;
object parent := nil;
backEdges is list of objects:= nil;
identifier GCI := nil;

procedure mark(object caller, color callerColor, identifier callerId)
returns color is
  if myColor = white then
     GCI := callerID;
     parent := caller;
     myColor := grey;
     for child in myChildren do
         childColor := child->mark(me, myColor);
         if childColor = black then
           myColor := black;
         endif
     od
  else if callerID not = GCI then
     return nil;
  else if myColor = black then
     return black;
  else if myColor = grey then
     backEdges->append(caller);
     return grey;
  endif

  if myColor = black then
    for neighbor in backEdges do
       neighbor->blacken();
    od;
  endif
  return myColor;
end;
```

Figure 5.3: Basic Algorithm, part 1

```
procedure blacken(identifier callerId) is
  if callerID not = GCI then
     return;
  endif;
  if myColor = grey then
    myColor := black;
    for neighbor in backEdges do
      neighbor->blacken();
    od
    parent->blacken()
  endif;
end;

procedure terminate(identifier callerId) is
  if myColor = white or callerId not = GCI
     then return;
  for child in myChildren do
     child->terminate()
  od
  if myColor = black then
    GCI := nil;
    myColor := white;
    parent := nil;
    backEdges := nil;
  else
    delete(me);
  endif;
end;
```

Figure 5.4: Basic Algorithm, continued

(or all back edges in the case of `blacken`) be traversed before returning to the parent. Therefore the `for` loops in lines 12, 28, 40, and 50 can all be executed in parallel, so long as they all terminate before the caller proceeds. In this case, some of the back edges may actually be cross edges.

A slight modification is also necessary to handle active objects, as neither the IRG nor the FRG is sufficient[29, 45]. If an active object is encountered, then it must traverse both its IRG and the FRG for which it is the root. This covers everything which refers to it, and everything it refers to.

5.5 Runtime of the Algorithm

The runtime of this algorithm depends entirely on the nature of the objects in the graph. We can measure this by considering the number of times an edge might be crossed, where each edge crossing is a message.

In an acyclic graph of only garbage, each IRG edge is traversed once, and the graph is collected on the way in. This gives a runtime of $O(2E)$.

In an acyclic graph with a PR or a cyclic graph containing only garbage, the `terminate` message is necessary, giving a runtime of $O(3E)$.

Finally, if there is a live cycle, it must be traversed an additonal time, yielding a worst-case runtime of $O(5E)$.

5.6 Fault Tolerance

As this algorithm is to be implemented in a network, it must be able to cope with a variety of network problems, from transient communication failures, to dead nodes, network partitions, and malicious messages.

Transient communication failures correspond to lost or repeated messages.

Our algorithm tolerates these so long as each message arrives once. Other than `terminate`, messages can only turn a *white* object to *grey*, or a *grey* object to *black*, so repeated messages cannot turn a *black* object to *grey*, or a *grey* one to *white*. `Terminate` messages are only sent after the traversal is finished, meaning that all messages for the traversal have arrived at least once. However they could cause the collection of live objects if they arrived during a later GCO. Therefore each GCO has a unique Garbage Collection Identifienr (GCI) based on the UI of the GCR. Messages with an inappropriate GCI are discarded.

Failed nodes and network partitioning require a more complex response, as some decision may need to be made about their permanence. Before considering difficult cases, some observations can be made:

- Partitions and failures are only significant when a GCO starts and can otherwise be ignored.

- If the traversal doesn't complete, no object is arbitrarily collected because of a failure or a partitioning.

- Parts of the IRG which do not encounter the failure or partition can continue. If a PR is found, then the problem can be ignored (by the algorithm).

Therefore, in many cases, the algorithm can complete partially and then be aborted.

Regarding other failures, our algorithm makes no claims to guarantee high availability. Nevertheless, it can support the sophisticated responses to faults necessary when such guarantees cannot be made. During execution of a GCO, all communication is between objects connected by an IRG edge. Problems manifest themselves along these same edges. We let the tail object of an edge in the IRG be responsible for determining the edge's state. This is very important if we consider that objects might consume considerable resources and expense. Let an edge, $(a,\ b)$ exist in the FRG, but the physical link has been severed. In a client-server model, a informs the server of its link to b, so b cannot become garbage. However, if b consumes or reserves large resources, its host may prefer to ignore a in determining if b is garbage (which may leave a with a dangling pointer if it is the only object refering to b, but that is a different problem).

The last issue to consider is correctness in the face of malicious or incompetent neighbors sending spurious messages. We assume pervasive access to encryption to positively identify the senders of messages. From the perspective of garbage collection, there are three kinds of error:

1. **A live object can be made to appear as garbage.** There are three ways this is possible:

   (a) A *black* object does not blacken its back edges.

   (b) A *black* object returns *grey* to a mark message.

   (c) An object sends a `terminate` message prematurely.

   Although we cannot prevent these, we can consider them to be the same as the *black* object removing its references to the *grey* object. We will treat it as such. Only the third case can actually cause an object to be collected. We can guard against this with two modifications:

   (a) Objects ignore the `terminate` message until they have finished searching their own subtrees.

   (b) Each object sends the `terminate` message to all its back edges. An object isn't collected until it has heard from each *grey* object encountered in the traversal. Unfortunately, this increases the worst-case runtime to $O(6E)$

2. **A garbage object is made to appear live, or an object is kept alive against the host's wishes.** The first condition is impossible by definition, but the second, denoting an object maintaining a reference just to keep another object from being collected, is quite possible in a client/server system, especially if it is possible to forge a reference. In our case, each object is aware of the references to it. It is also possible to periodically contact the possessor of a reference to verify its continued validity.

5.7 Multiple DGC operations

The algorithm is correct for a single GCO, but in actual practice there will be numerous concurrent, overlapping GCOs. Letting each operation proceed independently of all the others would produce correct results, so long as the objects can distinguish between GCOs, but would entail considerable duplication of efforts. If n interconnected nodes start garbage collecting at the same time, there would be $O(E*n)$ messages involved. This section will provide a modified algorithm to decrease the amount of repeated effort in overlapping GCOs, although there will remain a (hopefully rare) configuration in which the number of messages is still $O(E*n)$.

If GCOs are not to overlap, then when there is an encounter, one or the other, or both, must either block or retreat. Blocking and retreating each introduce their own potential problems.

1. The different GCOs might deadlock. If we consider each object as a resource the GCO needs to use, then we need to limit this access, leading to the

possibility of two GCOs waiting endlessly for each other.

2. GCOs might livelock, perpetually interrupting and restarting, but never completing.

3. Certain objects might derive erroneous information, and either garbage will not be identified, or live objects will be prematurely collected.

We will present an algorithm in which certain GCOs will block. Retreat is simpler and can be combined with a restart policy similar that used in CSMA/CD networks, such as Ethernet[57]. No performance bounds can be given for such an algorithm, but it might be quite practical in actual networks; a final judgement can only be given once real information is available about distributed object access patterns.

Our blocking algorithm will enforce two conditions to eliminate the problems mentioned above:

1. Where there is an encounter among GCOs, one operation will always be able to complete correctly.

2. Other traversals will block at the point of encounter until the privileged operation completes, ensuring that they do not propogate partial information.

We will enforce these conditions rather strictly here, although there are situations under which they can be relaxed.

We have already assigned to each GCO a unique key, based on the GCR's UI, to eliminate spurious `terminate` messages. Since these are enumerable, they can be ordered, yielding an ordering of GCOs (some hash function might be used to randomize this among objects). We will refer to this key as the *GCI*.

Suppose b is part of GCO_a with GCI_a and receives a `mark` message from c with GCI_x. There are three possible cases:

1. $GCI_x = GCI_a$. Object c is part of the same GCO, so everything proceeds as before.

2. $GCI_x < GCI_a$. Object c is part of a GCO of higher priority. Object b starts another traversal of its IRG as a member of GCO_x; b will not respond to its parent in IRG_a until GCO_x is completed. GCO_a is now blocked waiting on GCO_x.

3. $GCI_x > GCI_a$. Object c is part of a GCO of lower priority. Object b will not respond to c until GCO_a completes.

Case 2 assures completion of the GCO with the highest priority (no deadlock). Cases 2 and 3 assure that no GCO transmits partial information until the highest priority case terminates.

This algorithm is effectively a parallel sort of the GCOs. Depending on the order of encounters, the number of messages required varies from $O(n)$ (if encounters are in order of rising priority) to $O(n^2)$ (if they are in order of decreasing priority). Over the course of time, each GCO divides its neighbors into two groups, those of higher priority and those of lower priority. It waits until those of higher priority have sorted themselves (and terminated), terminates, and lets the lower priority neighbors sort themselves. In other words, this is a concurrent quicksort with an average of $O(nlogn)$ messages.

Security is difficiult to ensure in the case of concurrent GCOs because progress depends on the good will of objects claiming to be engaged in a different GCO. Coordinated lying among objects can easily lead to deadlock. Unlike lying in the case of a single GCO, coordinated lying cannot be easily reinterpreted as some form of acceptable behavior. At this point there are no obvious solutions. However, if there is reason to believe that an object is obfuscating, it is possible to demand sufficient information to show that there is an ongoing GCO of which it is a part. This would include producing a valid, fresh GCI, and naming the parents and children with whom it has communicated. In the final analysis, a GCO can be aborted and links to offending objects severed.

5.8 Controlling Cost

Although any single execution of the algorithm is linear in the number of edges crossed, and all garbage is eliminated in the first GCO to encounter it, the total cost to the network over time depends on how frequently it is run. In particular, the leaf nodes of non-cyclic garbage will be collected by the host nodes, so the greater the delay, the more likely that non-cyclic garbage will have collected itself, leaving only cycles to be collected. In this section we present show a minimum condition for an object to become garbage based on a "best guess" estimate of the nearest distance to a PR. We will also use these guesses to minimize the objects touched during a GCO.

We define the *Minimum Distance* (MD) of an object as the length of the shortest path through the FRG from an object to a PR. PRs have MD $= 0$. For other live objects this is one greater than the minimum MD among its parents in the FRG. An object is garbage only if its MD is infinite.

The MD is useful because it can only decrease if the object is alive. For an object's MD to decrease, there must be a new link created in its IRG, so it is alive. On the other hand, for the MD to increase, a link to a PR must be broken, an indication the object might now be garbage.

It is not feasible to constantly maintain the MD for every object, so we will use an approximation. Whenever an edge is formed in the FRG, we will mark it with the current MD of its tail object. We can also update these values during a GCO. Although the MD is now only an estimate, it can still only decrease by the formation of a new edge or participation in a GCO, and only increase by the removal of an edge. Therefore an object will only start a GCO when its MD increases. However, the object will traverse only one IRG child at a time, stopping when a PR is encountered.

Suppose we have some graph of objects rooted to some number of PRs, and assume for the moment there are no cycles. As the edges to those PRs are removed, the number of objects which can have an MD of 1 decreases. As that happens, the number of objects with MD of 2 must eventually decrease, etc. In other words, object MDs must eventually increase, triggering GCOs. When the last edge from a PR to some graph is removed, the objects will remove themselves in order.

To show this also holds for cycles, assume the cycle does not contain a PR (otherwise the whole cycle is alive). Then there must be some minimum MD among the objects in the cycle. This can only decrease if the cycle is alive. As in the acyclic case, every time a connection to a PR is removed, the MD of some object increases, triggering a GCO. When the cycle becomes garbage, this will trigger the GCO that removes it.

The above modifications arrange for garbage to be collected by increasing MD. This indicates it may not be necessary for an entire GCO to be performed; the traversal only need proceed to edges with lower MDs than the edge whose removal started the GCR, or objects are reached with no outgoing edges. We are considering a modification to this heuristic which should control the overhead of the algorithm.

Since non-cyclic garbage collects itself, we only need sufficient conditions to ensure that when a cycle becomes garbage, a GCO will occur that spans the entire cycle and shows that it is garbage. As we have shown above (again assuming there are no PRs in the cycle), at any given point in time before the cycle becomes garbage, there is a minimum MD> 0 along the cycle, and any objects with that MD are referred to by objects not on the cycle with a lower MD (or which had a lower MD at some point). Once the cycle has become garbage, one of these connections to the exterior of the cycle will eventually break, triggering one or more GCOs. Suppose each GCO only traverses the graph down any branch until it encounters an edge with lower MD than the removed one. This can lead to one of two outcomes:

1. As no such edge can exist between objects in the cycle (since the removed edge had a lower MD than any object in the cycle), if the cycle is not connected to any object outside itself, it will be collected.

2. If any such edges are encountered, they are outside the cycle. The traversal may change the MDs of all the objects in the cycle, but the cycle is still connected to something else, meaning that either it is not yet garbage, or the disappearance of other objects will trigger more GCOs inside the cycle.

We have not yet completed a rigorous analysis of the number of messages incurred by this heuristic. If we start to look at the amortized cost of garbage collection per distributed reference, there is the possibility of very reasonable performance. In the accounting, each distributed reference is allocated $O(1)$ messages for GCOs. The goal would be to apply the "credits" from self-collecting acyclic garbage to any excess traversals of edges between live nodes. An edge may be crossed any time the lowest MD edge to an FRG descendant is removed. However, for both the lower and higher MD edges to exist, some number of edges needed to have been created, which can be used to pay for this operation. We hope to show the amortized cost per live edge to be $O(1)$.

5.9 Implementation

Up to this point we have not discussed any actual implementation issues. In terms of data structures, we can reutilize the algorithms in Birrell et al.[5] and Piquer[44]. We should also be able to adapt the SSP Chains in Shapiro et al.[54]. The major local effort in producing an implementation will be establishing the IRG within each node, as this is normally not maintained. In the worst case this is $O(n^2)$ in the number of objects – for each incoming and outgoing FRG edge of the node, we need to determine if they are linked. However, if we are only interested in which incoming links are connected to any outgoing edge involved in a GCO, the problem is reduced $O(n)$, although intermediary objects need an extra flag to carry this information. This can be accomplished after a local GC by a DFS performed from the incoming links to the outgoing links traversing only objects not marked by the local GC. When an outgoing link involved in a GCO is encountered, that information is carried back up the tree. To keep from traversing edges more than once, objects need to be marked.

It is our intention to implement this algorithm shortly in the context of other work on cooperative applications in the Internet.

5.10 Conclusion

Garbage collection will be a necessary evil of mobile distributed systems. Here we present a bottom-up algorithm which eventually collects all the garbage present in the network without recourse to large-scale cooperation among processes. Individual DGCs can proceed in tandem. Where they overlap, a resolution mechanism is provided to minimize the number of extra messages necessary to coordinate among them.

This algorithm is a departure from existing DGC algorithms, in that it completely breaks with algorithms derived from non-distributed systems. Certainly, one wouldn't want to go the other way; the overhead of maintaining the inverse reference graph and other data structures required is prohibitive in a single process, but the incremental information necessary is much less when only distributed links are considered.

5.11 Current Status and Conclusion

At the current time, implementations exist of all the elements discussed here, with the exception of the distributed garbage collector, although I expect to undertake one soon; distributed garbage collection is essential to Dreme and any similar systems, but there is no system currently available for use in the Internet. These implementations support most of the functionality described in this document and have been sufficient to demonstrate the utility of these ideas for supporting distributed systems. I will continue working to improve this technology.

There have been two major events since I completed the first version of Dreme in early 1993 which will significantly affect future implementations. The first is the surge in popularity of the World Wide Web, and the second is the appearance of Sun's HotJava[16] browser.

Current Web technology, based on HTML, is utterly inadequate to support the information access needs of sophisticated clients, but has created a venue for hypermedia information to become generally available on the Internet. This was undoubtedly influential in my move from advocating a single language for describing interfaces to providing a system to support SGML in general. I am now extending the little SGML manipulation language described here to a full language with arbitrary Scheme code as the "semantic actions." It should generally handle DAGs of objects, to create SGML documents from other objects, as well as interpret and convert SGML to other forms. This will run in a programmable browser which can accept or send documents via http, SNMP, local file system, or Dreme objects.

Although Java is less powerful than Dreme as a distributed programming language, it, too, requires the wide availability of a byte-code interpreter running on a variety of platforms. The impending success of HotJava as the immediate next generation of Web browser presents the opportunity for a new implementation of Dreme to take advantage of the wide availability of Java interpreters. The key to doing so is rewriting Dreme to use the Java VM. If that is done, particularly if Dreme code is able to communicate with Java objects as it currently does with C++ objects, then Dreme objects will be able to migrate to any Java interpreter and communicate with the user there. The difficult element of this effort is mapping Dreme's CPS architecture to the Java VM, which is based on a traditional stack architecture.

Programming in a world of vast distributed information sources and transformers will be qualitatively different than our current procedures. The goal of this dissertation is to provide some assistance getting there

Bibliography

[1] Gul A. Agha. *Actors*. MIT Press, 1986.

[2] G. T. Almes et al. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, 11(1):43–58, 1985.

[3] Tim Berners-Lee and Daniel Connolly. Hypertext Markup Language: A representation of textual information and metainformation for retrieval and interchange. Technical report.

[4] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE '87*, pages 176–187. Springer Verlag LNCS 259, 1987.

[5] Andrew Birrell et al. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 1993.

[6] Andrew Black. Supporting Distributed Applications: Experience with Eden. *Operating Systems Review*, 19(5):181–193, 1985.

[7] Andrew Black and Yeshayahu Artsy. Implementing location independent invocation. In *9th Int. Conf. on Distributed Computing Systems*. IEEE, 1989.

[8] Jean-Pierre Briot and Jean de Ratuld. Design of a distributed implementation of ABCL/1. *SIGPLAN Notices*, 24(4):15–17, 1989.

[9] Luca Cardelli. Obliq: a language with distributed scope. Technical Report 122, DEC SRC, 1994.

[10] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–60, 1995.

[11] David Chaiken. Cache coherence protocols for large-scale multiprocessors. Master's thesis, MIT, 1990.

[12] William Clinger, Jonathan Rees, et al. Revised ^{4} Report on the Algorithmic Language Scheme. Technical report.

[13] H. Corporaal. Distributed heapmanagement using reference weights. In *Distributed Memory Computing*, pages 325–336. Springer Verlag LNCS 487, 1991.

[14] Daniel Friedman and Mitchell Wand and Christopher T. Haynes. *Essentials of Programming Languages.* MIT/McGraw-Hill, 1992.

[15] Benjamin Goldberg. Generational reference counting. In *Conference on Programming Language Design and Implementation*, pages 313–321. ACM, 1989.

[16] James Gosling and Henry McGilton. The java language environment. Technical report, Sun Microsystems Computer Company, 1995. ftp://ftp.sun.com/docs/JavaBook.ps.tar.Z.

[17] Christopher Haynes and Daniel Friedman. Engines build process abstractions. In *Symposium on Lisp and Functional Programming*, pages 18–24. ACM, 1984.

[18] Dan Heller. *MOTIF Programming Manual.* O'Reilly and Assocs., 1994.

[19] Carl E. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1972.

[20] Carl E. Hewitt et al. Actor Induction and Meta-evaluation. In *Symposium on Principles of Programming Languages*, pages 169–182. ACM, 1973.

[21] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[22] Yuuji Ichisugi and Akinori Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In T. Ito and A. Yonezawa, editors, *Concurrency: Theory, Languages, and Architectures*. Springer Verlag, 1989.

[23] Williams Ludwell Harrison III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, 2(3):179–396, 1989.

[24] ISO/IEC. *DIS 10179.2 Document Style Semantics and Specification Language.*

[25] ISO/IEC. *IS 10744 Hypermedia/Time-based Structuring Language (HyTime)*, 1992.

[26] James E. White. Telescript technology: The foundation of the electronic martketplac. Technical report, General Magic, 1994.

[27] James Gosling and David S. H. Rosenthal and Michelle J. Arden. *The NeWS Book*. Springer-Verlag, 1989.

[28] Eric Jul et al. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[29] Dennis Kafura et al. Garbage collection of actors. In *OOPSLA/ECOOP Proceedings*, pages 126–134, 1990.

[30] R. Kessler et al. Implementing concurrent scheme for the mayfly distributed parallel processing system. *Lisp and Symbolic Computation*, 5:73–93, 1992.

[31] Robert Kessler and Mark Swanson. Concurrent scheme. In *Parallel Lisp: Languages and Systems*. Springer Verlag LNCS 441, 1990.

[32] Guy L. Steele, Jr. *Common Lisp, the Language, 2nd Edition.* Digital Press, 1990.

[33] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *12th Int. Conf. on Distributed Computing Systems*, pages 708–715. IEEE, 1992.

[34] Bernard Lang et al. Garbage collecting the world. In *19th Symposium on Principles of Programming Languages*, pages 39–50. ACM, 1992.

[35] H. M. Levy and E. D. Tempero. Modules, objects and distributed programming: Issues in RPC and remote object invocation. *Software Practice and Experience*, 21(1):77–90, 1991.

[36] Henry Lieberman. Concurrent object-oriented programming in Act 1. In *Object Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[37] Marc Linton. Fresco tutorial. ftp://ftp.sgi.com/graphics/fresco/exug94.ps.Z, 1994.

[38] Marc Linton and Chuck Price. Building distributed user interfaces with fresco. Technical report, Silicon Graphics, 1993. ftp://ftp.sgi.com/graphics/fresco/xconf93.ps.Z.

[39] Brad Myers, editor. *Languages for Developing User Interfaces.* Jones and Bartlett Publishers, 1993.

[40] Brad Myers. Why are human-computer interfaces difficult to design and implement. Technical Report CMU-CS-93-183, Carnegie Mellon University, 1993.

[41] Niels Juul and Eric Jul. Comprehensive and Robust Garbage Collection in a Distributed System. In Y. Beckers and J. Cohen, editor, *International Workshop on Memory Management.* LNCS 637, 1992.

[42] Oscar. M. Nierstrasz. Active Objects in Hybrid. In *OOPSLA Proceedings*, pages 243–253, 1987.

[43] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[44] Jose M. Piquer. Indirect reference counting. In *PARLE '91*, pages 150–165. Springer Verlag LNCS 505, 1991.

[45] Isabelle Puaut. Distributed Garbage Collection of Active Objects with no Global Synchronisation. In *International Workshop on Memory Management*, pages 148–164. Springer Verlag LNCS 637, 1992.

[46] Isabelle Puaut. A Distributed Garbage Collector for Active Objects. In *OOPSLA Proceedings*, pages 113–128. ACM, 1994.

[47] Christian Queinnec. A concurrent and distributed extension of scheme. In *PARLE '92*. Springer Verlag, 1992.

[48] Christian Queinnec. *Les langages lisp*, chapter 11. InterEditions, 1994.

[49] Christian Queinnec. Locality, Causality, and Continuations. In *Symposium on Lisp and Functional Programming*, pages 91–102. ACM, 1994.

[50] R. K. Raj et al. Emerald: A general-purpose programming language. *Software Practice and Experience*, 21(1):91–117, 1991.

[51] John H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, 1992.

[52] Jr. Robert Halstead. Implementation of Multilisp. In *Symposium on Lisp and Functional Programming.* ACM, 1984.

[53] B. Schneiderman. *Designing the User Interface.* Addison-Wesley, 1979.

[54] Marc Shapiro et al. SSP Chains. In *Symposium on Principles of Distributed Computing*, pages 135–146. ACM, 1992.

[55] Etsuya Shibayama and Akinori Yonezawa. Distributed computing in ABCL/1. In *Object Oriented Concurrent Programming*, pages 91–128. MIT Press, 1987.

[56] Mark R. Swanson. Concurrent scheme reference. *Lisp and Symbolic Computation*, 5:95–104, 1992.

[57] Andrew S. Tannenbaum. *Computer Networks, 2nd Edition.* Prentice Hall, 1989.

[58] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[59] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE '87*, pages 432–443. Springer Verlag LNCS 259, 1987.

[60] Paul Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas.

[61] Paul Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, pages 1–42. Springer Verlag LNCS 637, 1992.

[62] Akinori Yonezawa et al. Modelling and programming in an object-oriented concurrent language ABCL/1. In *Object Oriented Concurrent Programming.* MIT Press, 1987.