The Object-Oriented Components of the Enterprise Parallel Programming Environment

Greg Lobe, Duane Szafron, Jonathan Schaeffer
Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1
{greg, duane, jonathan}@cs.ualberta.ca

ABSTRACT

The *Enterprise* programming environment supports the development of applications that run concurrently on a network of workstations. This paper describes the object-oriented components of *Enterprise*, implemented in Smalltalk-80, and their seamless integration with the procedural components, implemented in C. The object-oriented user-interface supports a new anthropomorphic model for parallel computation that eliminates much of the perceived complexity of parallel programs. The object-oriented animation component is a new animation architecture that supports synchronous and asynchronous events. This allows a user to view the dynamic interactions of the parallel components of a distributed application to simplify performance monitoring and debugging. The *Enterprise* experience highlights the strengths of object-oriented methodologies both for expressing user models and for implementing related components.

1. Introduction

This paper describes how object-oriented techniques were used to design and implement components of the *Enterprise* programming environment that supports the development of distributed applications for networks of workstations. *Enterprise* is a good example of an embedded application where object-oriented and traditional code co-exist. Object-orientation was used in the design of the parallel programming model and Smalltalk-80 (ST-80) was used in the implementation of the user-interface and program animation components. The rest of the system was written in C.

Parallelism adds an extra dimension of complexity to the design, implementation, and debugging of programs. When multiple processes run on multiple processors (dozens, hundreds or more), the user often has difficulty understanding a parallel computation using conventional sequential tools. Visualization and animation are needed to grasp the often intricate and non-deterministic interactions between processes. More importantly however, a simple model is needed to bring order to an often chaotic collection of asynchronous processes.

In *Enterprise*, the interactions of processes in a parallel computation are described using an analogy based on the parallelism in a business organization. Since business enterprises efficiently coordinate many asynchronous individuals and groups, the analogy is beneficial to understanding and reducing the complexity of parallel programs. Inconsistent parallel terminology (master-slave, pipelines, divide-and-conquer, etc.) is replaced with more familiar business terms (*assets* called *departments*, *receptionists*, *individuals*, *divisions*, *representatives*, etc.). Every sequential procedure that will execute concurrently is assigned an asset type that determines its parallel behavior. The user code for each of these procedures is sequential C, but a procedure call to such an asset is automatically translated to a message send by *Enterprise*.

Consider the following user C code, assuming that *func* is an asset in the program:

```
result = func( x, y );
/* other C code */
a = result;
```

When *Enterprise* translates this code to run on a network of workstations, the parameters `x` and `y` are packed into a message and sent to the

This is a pre-print of a copyrighted article in Technology of Object-Oriented Languages and Systems Conference 11, August, 1993, pp. 215-229.

process that executes the asset `func`. The caller continues executing and only blocks and waits for the function result when it accesses the result`(a = result)`. *Enterprise* also supports passing parameters by reference.

Enterprise has three components: an object-oriented graphical interface, a pre-compiler, and a run-time executive. The user specifies the application parallelism by drawing a hierarchical *enterprise* that consists of assets. At run-time, each asset corresponds to a process. Sequential procedure calls in C are translated into message send/receives across a network by the pre-compiler. The execution of the program (process/processor assignment, establishing communication links, monitoring network load) is done by the run-time executive. More information about *Enterprise* including the anthropomorphic programming model, the system implementation and a user appraisal is in [LMP92] and [Par93].

The graphical interface and the *Enterprise* anthropomorphic model are used for program design. However, they can also used be used to monitor or replay an execution. The interface animates the states of the assets (processes) and the messages that are sent between them. These facilities are currently being expanded to include performance monitoring and debugging.

This paper describes the design of the *Enterprise* interface and its animation capabilities. Several object-oriented research contributions and lessons were derived:

1. a new anthropomorphic model for parallel computation,
2. an application-independent object-oriented animation architecture,
3. a technique for integrating object-oriented software with non-object-oriented software,
4. evidence that multiple inheritance is essential for the proper representation of those object-oriented applications that depend on real-world models or analogies,
5. how object-oriented techniques can be used in software development environments that support non-object-oriented languages, and
6. how context-sensitive hierarchical direct manipulation user-interfaces can simplify user models, focus user attention and prevent errors.

2. Using Enterprise

This section presents an example of how *Enterprise* is used to construct a distributed program. Consider a *Simulation* program that displays a group of fish swimming across a display screen. This problem was contributed by a research group in our Department and is more complex than portrayed by the following description. The main procedure, *Model*, consists of a loop that, for each frame in the simulation, performs some work on the frame and calls *PolyConv*. *PolyConv* manipulates the image received from *Model* and calls *Split*. *Split* polishes the frame and writes it to disk.

An *Enterprise* user manipulates icons that represent high-level program components called *assets*. An asset represents a single C function, called an *entry procedure*, together with a collection of support procedures used by the entry procedure. A program will consist of several assets. In this example, there will be three assets: *Model*, *PolyConv* and *Split*.

Initially, the *Enterprise* window contains one view called the *Enterprise View*. It contains the icon for one *enterprise* asset that represents a new program. Each asset has a context-sensitive pop-up menu. For example, if the user selects *Name* from the *enterprise* menu and types the word *Simulation* into a dialog box, the *enterprise* would be named *Simulation* as shown in Figure 1.

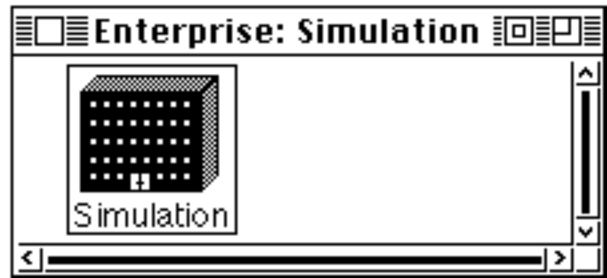

Figure 1: The *Simulation* program

The user-interface is implemented in ST-80 which uses the host windowing system. The figures in this report were generated on a Macintosh and look similar with X windows.

If the user then selects *Expand* from the asset menu, the *enterprise* icon will expand to reveal the single *individual* that it contains. To name this asset, the user selects *Name* from the

asset menu of the *individual* and types the word *Model* into the dialog box that appears.

The user could enter all of the code for *Model*, *PolyConv* and *Split* into this *individual* and run the program sequentially. However, there is no reason why *Model* should wait until *PolyConv* completes the first simulation frame to start processing the second frame. Similarly, *PolyConv* does not need to wait for *Split*. In the parallel processing community this type of parallelism is often called a pipeline.

Using the *Enterprise* analogy, these three routines act like an assembly or production line and are represented by a *line*. Therefore, if the user selects *Line* from the asset menu of *Model*, it is re-classified as a *line*. After re-classification, the *individual* appears as a *line* consisting of a *receptionist* and one subordinate *individual*. Figure 2 shows the *line* where the numeral 1 indicates the number of subordinate assets.

If the user selects *Expand* from the asset menu of *Model*, it is expanded to reveal its two components. Since three components are required, the user selects *AddAfter* from the last component's menu to add a third asset and names the new assets, *PolyConv* and *Split*. The user then selects *Code* from each asset menu in turn and enters C source code into text editor windows, as shown in Figure 3.

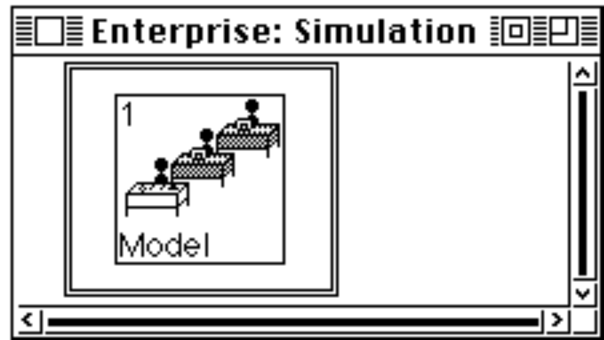

Figure 2: A line in the *Simulation* program

The double line rectangle represents the *enterprise*. The dashed-line rectangle represents the *line* and each inner icon represents a component. The first component is a *receptionist* that shares the name, *Model*, with the *line* that contains it. All calls to a *line* are received by the *receptionist*. The other two components are subordinate *individuals*.

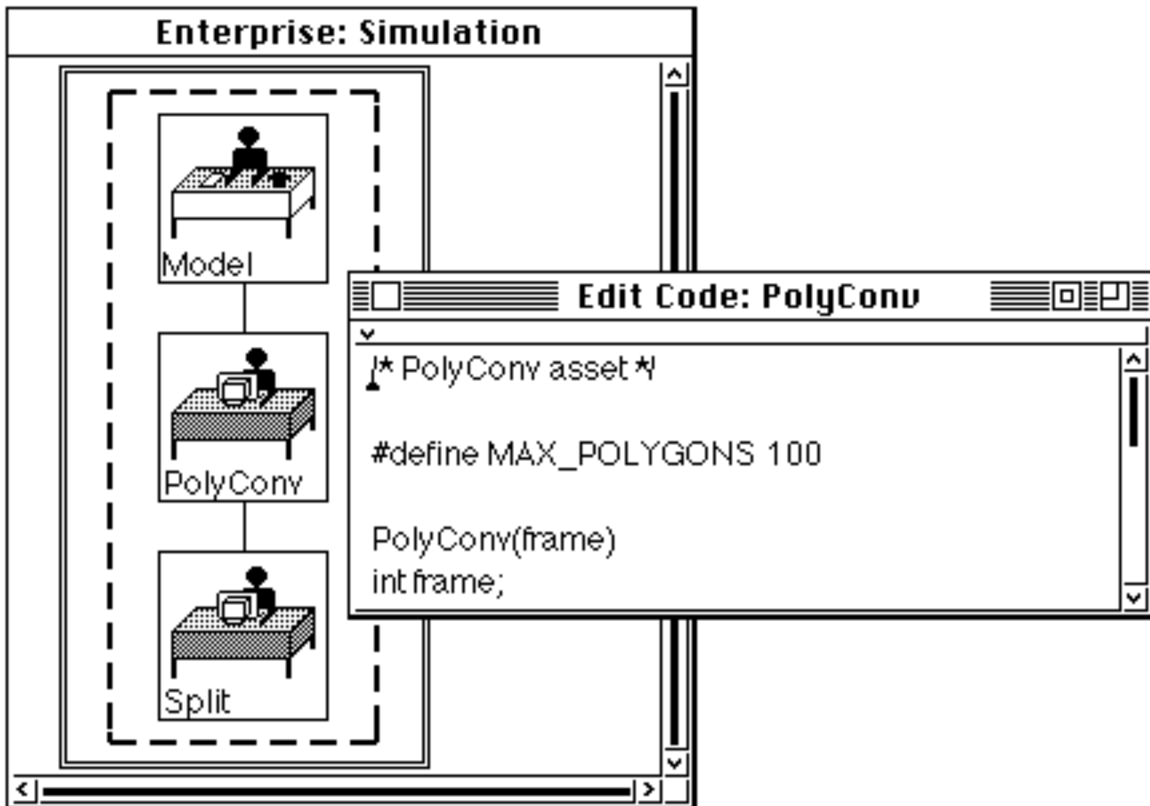

Figure 3: Editing the C source code.

If the user selects *Compile* from the *Enterprise* view menu then the *Enterprise* pre-compiler automatically inserts code to handle the distributed computation, compiles the program and reports any errors in a window. Once the program is compiled, the user selects *Execute* and *Enterprise* finds as many processors as are necessary to start the program and initiates processes on the processors.

One of the strengths of the *Enterprise* model is that it is easy to experiment with alternate parallelization techniques without changing C source code. Each asset represents at least one process. If a call is made to the *individual Split*, it is executed by a process and if a subsequent call is made to *Split* before the first call is complete, the second call must wait for the first call to finish.

However, if the *Split* asset is *replicated* then multiple processes can be used to execute calls concurrently. For example, if the user selects *Replicate* from the asset menu of *Split* and enters 1 and 5 as minimum and maximum replication factors in the dialog box that appears, then *Split* is replicated as shown in Figure 4.

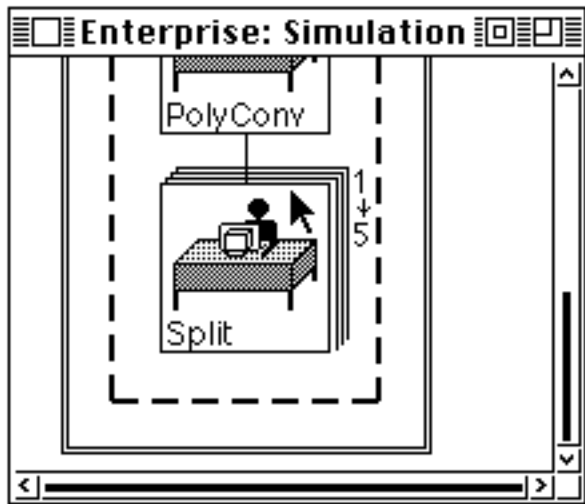

Figure 4: A replicated asset.

When *PolyConv* calls *Split*, a process is initiated and if a subsequent call is made to *Split* before the first call is done then a second process is initiated (if there is an available machine). Replication can be dynamic in *Enterprise* so that as many processors as are available on the network may be used, subject to a lower and upper bound supplied by the user. Several other asset kinds are supported by *Enterprise* and they can be combined in arbitrary hierarchies.

3. The User-Interface Implementation

The *Enterprise* user-interface has been implemented in ST-80, version 4.0. It may be used to construct programs on any machine that supports ST-80 including a broad range of Unix workstations as well as Macintoshes and IBM X86 or compatible machines. However, since the rest of the *Enterprise* programming environment is Unix dependent, features such as *Compile* and *Execute* only work on Unix workstations where the ST-80 interpreter runs as a single task under X windows.

The history of the *Enterprise* user-interface is an interesting one and illustrates some of the tradeoffs that can occur when deciding whether or not to use object-oriented technology and how to integrate it with an existing software legacy. *Enterprise* is based on a predecessor programming environment called *Frameworks* [SSG91] that was completely implemented in C. The *Frameworks* environment had a primitive graphical user-interface that lacked the anthropomorphic model and required the user to do more drawing. When the *Enterprise* project was started, a decision was made to create an object-oriented graphical user-interface that could more easily represent the new high-level parallel programming model.

Since the researchers had some experience with the object-oriented languages, Smalltalk and C++, both were considered for implementing the user-interface. C++ was chosen for three reasons: it has faster run-time performance than Smalltalk, it should be easier to integrate a C++ user-interface with existing C code since it is a superset of C and, unlike ST-80, there are no licensing restrictions on the distribution of a C++ user-interface. Smalltalk/V was disqualified since it does not currently run under Unix.

The Interviews [LVC89] user-interface class library was used to reduce development time. Unfortunately, 6 person-months were spent trying to implement the user-interface using Interviews without success. Although individual widgets were relatively easy to build, the complexity of Interviews resulted in a

learning curve that was too steep. Although an experienced Interviews programmer may have been able to complete the task in this time, our programmer could not.

Since the user-interface was lagging behind the pre-compiler and executive, we then decided to try Motif [You92]. However, two person-months of work on Motif (by a different programmer) yielded results that were no better.

At this point, we decided to try ST-80 in spite of its perceived problems. A graduate student who had previously taken a one semester course in object-oriented computing that included ST-80 as a component then produced a working prototype of the user-interface in three weeks! Of course the final user-interface (with animation) as described in this paper took much longer (about four months). The execution speed of the user-interface is well within our performance requirements and it was quite easy to integrate the ST-80 user-interface with the C pre-compiler and executive. The rest of this section describes the way the user-interface was implemented in ST-80.

3.1 The User-Interface Control Model

Since a program may display many ST-80 windows, the ST-80 interpreter polls the windows, asking each in turn if it wants control. The default behavior is that a window takes control whenever the cursor is inside of it. The Model View Controller (MVC) paradigm [LP91] is used where the model is an instance of class *Enterprise*, the view is an *EnterpriseWindow* and the controller is an *EnterpriseController*. The *EnterpriseController* behaves exactly the same as a default *Controller* except when the program is animated and this will be described in Section 4.

The model is responsible for knowing its *enterprise* (program). The window is responsible for displaying the *enterprise* using the values stored by the model. Views are composite objects that can contain sub-views, but the location and size of a sub-view within its parent view is maintained by a wrapper object. That is, sub-views are contained in wrappers, which are themselves contained in a parent view. An instance of *EnterpriseWindow* contains two wrapped sub-views, an *Enterprise* view and a *Service* view. The *Enterprise* view displays the *enterprise* (program) and the *Service* view displays the *service* assets used by the *enterprise*. The *Service* view can be hidden when it is not used. *Service* assets are described in [LMP92].

When a mouse button is pressed, the window passes control to the view that contains the cursor. The view then determines which asset (if any) was selected. The selected asset is one whose bounds (rectangle) contains the cursor point. However, since assets may be nested in a hierarchical structure, many assets may contain the cursor point. The selected asset is defined as the innermost one that contains the cursor point. For example, in Figure 4, the cursor is inside of the *individual Split*, which is inside the *line* (dashed line) *Model*, which is inside the *enterprise* (double line) named *Simulation*. In this case the cursor point is considered to be inside *Split*.

If an asset is selected, a context-sensitive menu is displayed that contains only the operations that are valid for the selected asset. For example, if an asset is expanded, then the *Collapse* operation would appear in the menu, but the *Expand* operation would not. This makes it impossible for a user to select an invalid operation. If no asset is selected, then the menu for the *Enterprise* view is displayed.

This approach simplifies the user's mental model of the programming environment since it reduces the number of operations the user sees [LSW87]. It is in stark contrast to pull-down menus where the user is presented with a plethora of choices some of which have subtle differences and some of which do not even apply to the user-interface component being considered. For example, if the user chooses *Compile* from an asset's menu, only the code for the asset is compiled. If the user chooses *Compile* from the *Enterprise* view menu, then all assets are compiled. Furthermore, the *Execute* command does not even appear in an asset menu. In a pull-down system, *Compile Asset*, *Compile Program* and *Execute* would all appear in the menus.

How does a view determine which of its assets is selected? A traditional non-object-oriented approach would be for a view to maintain a list of its assets and their locations and to compute the selected asset based on this information. However, since assets can be

nested, some other structural information would be required as well. Assets can be expanded to reveal their components or collapsed to hide their internal details. As assets are expanded and collapsed, their locations change and must be updated. In the object-oriented world, each asset should be responsible for knowing its own location and its structure (its parent asset and the other assets it contains). The view itself only needs to know the *enterprise*.

When the *enterprise* or any other asset is passed the cursor point and asked for the selected asset, it behaves recursively as follows. If the point is outside its bounds it answers *nil*. If the point is inside its bounds and it does not contain any component assets or it contains component assets but they are not currently displayed, then it returns itself. Otherwise, the asset asks each of its component assets in turn to identify the selected asset until one answers an asset or all respond with *nil*. The asset then returns this result. Before asking each component asset, the asset asks the wrapper of the component to change the coordinates of the cursor point to the local coordinates of the component.

3.2 Drawing Assets

When an asset receives a display message, it draws itself. Any asset that contains component assets can be either collapsed or expanded. Assets that are collapsed or do not have components are displayed in the same way. The asset draws its icon and displays its name in the lower left corner of the icon. If the asset is replicated, lines are drawn above and to the right of the icon to simulate a stack of icons and the number of replications is displayed outside of the top right corner of the icon.

An expanded asset first draws a rectangular border. The size of the rectangle is computed by asking each component for its size and adding room for space between the components. Next a display message is sent to each component so that it draws itself. The parent asset then draws the connections between the components. Finally the replication is indicated in the same way as it is for collapsed assets.

The basic drawing behavior is implemented in the *Asset* class and each *Asset* subclass provides a method for drawing its own icon. In addition, different line styles are used for the borders of expanded assets. For example, *enterprise* assets use two lines separated by one pixel, *line* assets use a dashed double width line, and *division* assets use a double width wavy line. The method that draws the border is overridden in these assets to use the appropriate behavior. Similarly, the method that draws connections is overridden to draw the correct connections for the various *Asset* sub-classes.

3.3 The Other Enterprise Components

Although the user-interface is implemented in ST-80, two other *Enterprise* components are implemented in C. The user-interface communicates with the pre-compiler and the executive through Unix pipes and text files. This section describes the technique for connecting to the external Unix processes, the organization of the directories containing C source and object code files for a program, and three other kinds of text files that are used to communicate with the other components.

<u>Graph, Event and Preference Files</u>

A graph file describes a single *Enterprise* program. It specifies the hierarchical structure of the assets, replication factors, compile and link options, and any user machine preferences. The assets are listed in a depth-first order. For each asset there is a line with its name, type, replication factor and options for ordering, debugging and optimization. If the asset has internal components there is also a count of components. Following this are four lines that specify the compile, link and run options. If the asset has components, these lines are followed by the description of the components in the same format.

Graph files are created and edited by the user-interface. When the user selects *Save*, *Compile*, or *Run* from the *Enterprise* view menu, the *enterprise* stores a representation of itself in a graph file whose name is the *enterprise* name with a ".graph" appended. Each asset type knows how to write a description of itself and if it has components, it asks its components to write themselves as well. Alternately, when the user wants to load a previously saved program, the graph file is read and as it is parsed, assets are created and displayed. The pre-compiler uses a program's graph file to identify procedure/function calls to

assets and replaces them with message sends and receives. The run-time executive uses the graph file to determine how many processes to launch, the execution role of each process and the appropriate communication links between these processes.

Event files are created by the run-time executive's monitor process while a program is running and are used later to animate the program. The events they contain are described in more detail in Section 4.

Enterprise maintains a preferences file. When the user-interface first starts, it looks in the current directory for a file named *.entrc*. If the file exists, it is read and global preferences are set from its contents. For example, the user's text editor is specified by a line of the form *EDITOR= editor name*.

Enterprise Directories for Source Code

When a new program is created, *Enterprise* creates a new sub-directory with the same name as the program. It then creates sub-directories of this directory to organize the files used by the program. The directories are: Assets (C source code for assets), User (C source code for internal asset procedures), Include (header files), Out (input and output files), Obj (object files for each asset), Bin (executables), Sys (*Enterprise* generated files) and Src (pre-compiler output).

External Processes

The user-interface launches external processes for compiling code, running a program and (possibly) for editing code. The user may use a standard ST-80 editor or, under Unix, a non-ST-80 editor may be selected. Several editors can be active at the same time (one for each asset). If the ST-80 editor is used, no new process is launched. Instead, a new ST-80 window is created and the window is added to the list of active windows. It is given control by the ST-80 interpreter whenever its window has the cursor. If an external editor is used, an X window is created. The editor becomes an X window's task that executes concurrently with the ST-80 interpreter.

The *Compile* and *Run* commands are only usable with the Unix version of the user-interface since the pre-compiler and executive currently require Unix. Both commands launch an external process and establish communications with it. ST-80 simplifies this task by providing a *UnixProcess* class. A message is sent to this class specifying the name of a Unix program, an array of arguments for the command and a block. The block is evaluated with the external process as an argument. This provides a mechanism for referencing the process from ST-80 after it has been created. When the message is sent, the process is created and two pipes are established, one connected to the process' standard input and the other connected to both its standard output and standard error. These pipes are represented as ST-80 streams that are contained in the instance of *ExternalConnection* that is returned by the message.

The user can elect to compile and link the entire program or to compile part of the asset hierarchy. In either case, if the program has been changed, the user-interface first writes out the graph file. The *Enterprise* pre-compiler process is then started and a window is created to display all text that is sent to the *External Connection*'s output stream. The *Enterprise* view's controller monitors the stream. Whenever new text is available, it is displayed in this window. If there is no new text, the polling loop just continues normally. The user can interact with the system normally and may even cancel the compile. When the compile is finished, the window is left open so that the user can review the compiler messages. Programs are run in a similar manner except output is displayed in another window.

3.4 The Asset Inheritance Hierarchy

Section 3.2 described the way that assets are drawn and the approach relied heavily on inheritance. In fact, inheritance is used extensively throughout the user-interface, but the asset hierarchy can be used to illustrate its importance. The asset kinds form a natural inheritance graph as shown in Figure 5.

A solid triangle in the upper left corner of a class denotes an abstract superclass as described in [WWW90]. The abstract class *Asset* is the root of the inheritance tree. Universal responsibilities like naming are defined and implemented in this class.

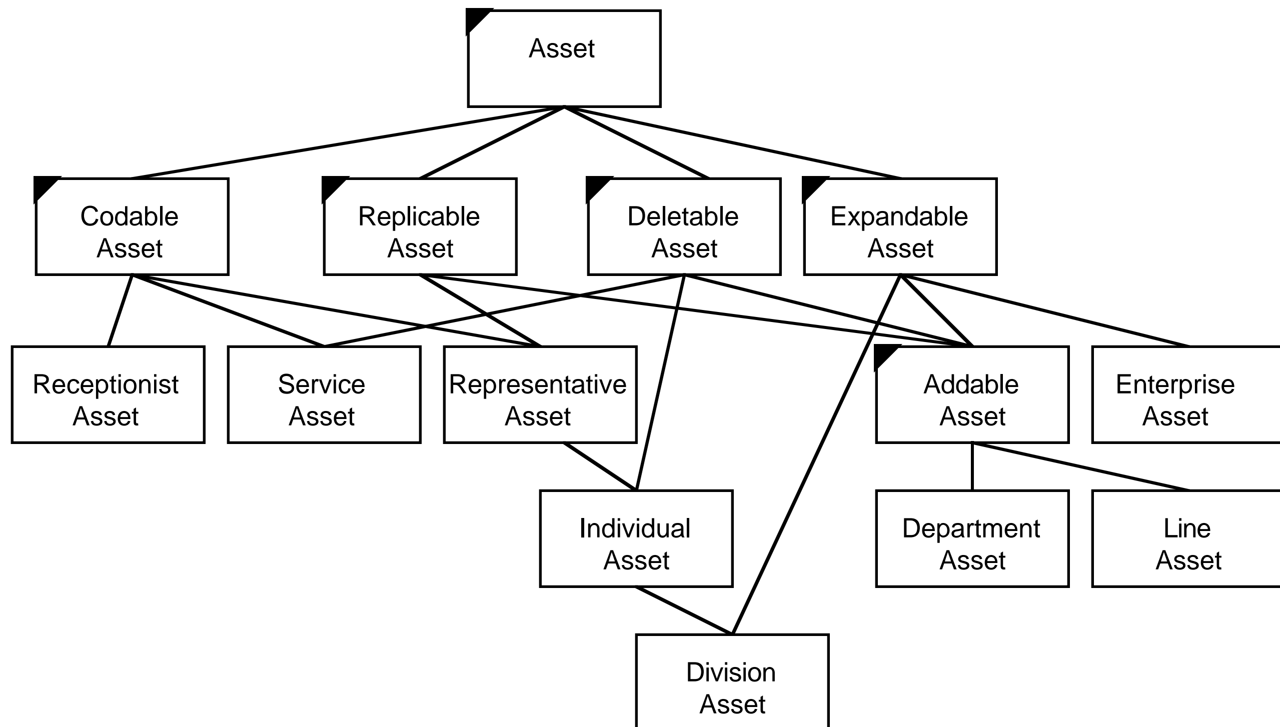

Figure 5: The asset inheritance graph.

Below the *Asset* class is a level of abstract superclasses that define several responsibilities that are shared by several of the leaf asset classes. A *CodableAsset* has an external file of C source code associated with it which can be edited and compiled. A *ReplicableAsset* can be replicated and transformed to an asset of a different type. A *DeletableAsset* can be deleted from its parent asset. An *ExpandableAsset* has component assets so it can be expanded or collapsed. An *AddableAsset* can have components added to it after it has been created.

The rest of the asset classes are concrete subclasses. A *ReceptionistAsset* has code, but can't be replicated, deleted, or expanded. A *RepresentativeAsset* has code and can be replicated but can't be deleted or expanded. An *IndividualAsset* is like a *RepresentativeAsset*, except that it can be deleted. A *DivisionAsset* is like an *IndividualAsset*, except that it can be expanded. A *ServiceAsset* has code and can be deleted, but it can't be replicated or expanded. A *LineAsset* or *DepartmentAsset* can be replicated, deleted, or expanded, but has no user code. An *EnterpriseAsset* is expandable, has no user code, can't be replicated and can't be deleted.

Unfortunately, ST-80 is restricted to tree inheritance so several compromises were made in transforming this inheritance structure to a tree. The result is shown in Figure 6. A comparison of Figures 5 and 6 illustrates clearly that support for multiple inheritance is essential for applications with real-world models. The lack of multiple inheritance was the most difficult obstacle that needed to be overcome in using ST-80 for the *Enterprise* project.

ReplicableAsset and *DeletableAsset* were merged with *Asset*. The rounded rectangles contain the main messages defined by each class and the symbol ~ means that a message was overridden because it should not exist for a class. For example, the *ReceptionistAsset* class overrides the replicate, coerce, and delete methods. The *Division* class was made a subclass of *ExpandableAsset* instead of *IndividualAsset*. The methods for editing code were then re-implemented in *DivisionAsset*. In addition to these changes, the *Asset* class itself was made a subclass of the ST-80 pre-defined class *CompositeView* so that all assets could inherit the behavior of visual objects that have sub-parts.

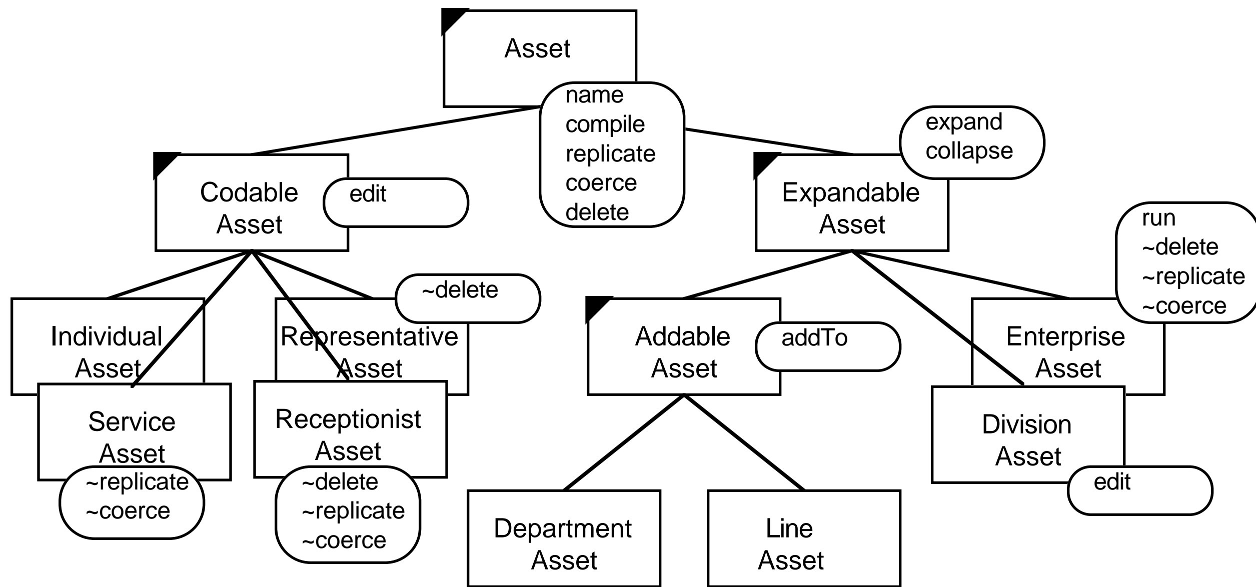

Figure 6: The asset inheritance tree.

4. Program Animation

Enterprise program animation is used to monitor a program's performance and to identify parallel programming and logic errors at the message (asset) level. The user can examine the amount of parallelism, when and where synchronization occurs, which machines are being used and their load, the lengths of message queues, and the state of each process during execution. Currently, there are no debugging facilities for setting breakpoints or examining the values of variables. Animation consists of displaying asset states, displaying messages as they move between assets and displaying message queues.

Enterprise replays execution of a program using an event file that was produced during execution. The event file is produced by an external Unix process that receives messages from the *Enterprise* executive process and logs events to the file. The interface assumes that the events are partially ordered [Lam78] so it is the responsibility of the executive process or a post-processor to do this. To support real-time animation, it is possible to replace this file by a stream connection between the user-interface and event-monitoring processes. However, in this case, the animation system may be unable to keep up with events. Therefore, replay is the preferred approach to animation.

During animation, the time between animation steps is proportional but not equal to real time. The proportionality factor can be adjusted by the user to adjust the speed of the animation. The user can also step through the animation one event at a time.

4.1 Animation View

When the user selects *Animate* from the *Enterprise* view menu, the *Enterprise* view is replaced by an *Animation* view. Each replica from a replicated asset is displayed as a separate icon and named by appending an id number to the base asset name. For each asset, the id numbers are generated in sequential order starting at 1. Messages and message queues are displayed as icons and animation commands appear in the asset, message queue and *Animation* view menus. For example, the user can use an asset menu to open a monitoring window that contains such information as the machine name for the asset and performance information for that machine. Similarly, the user can use the message queue menu to examine the details of messages that it contains. Finally, the view menu can start the animation from the beginning, pause or resume the animation, single step through events, set the speed of the animation and replace the *Animation* view by the *Enterprise* view. The *Animation* view of a recursive *AlphaBeta* tree search program is shown in Figure 7.

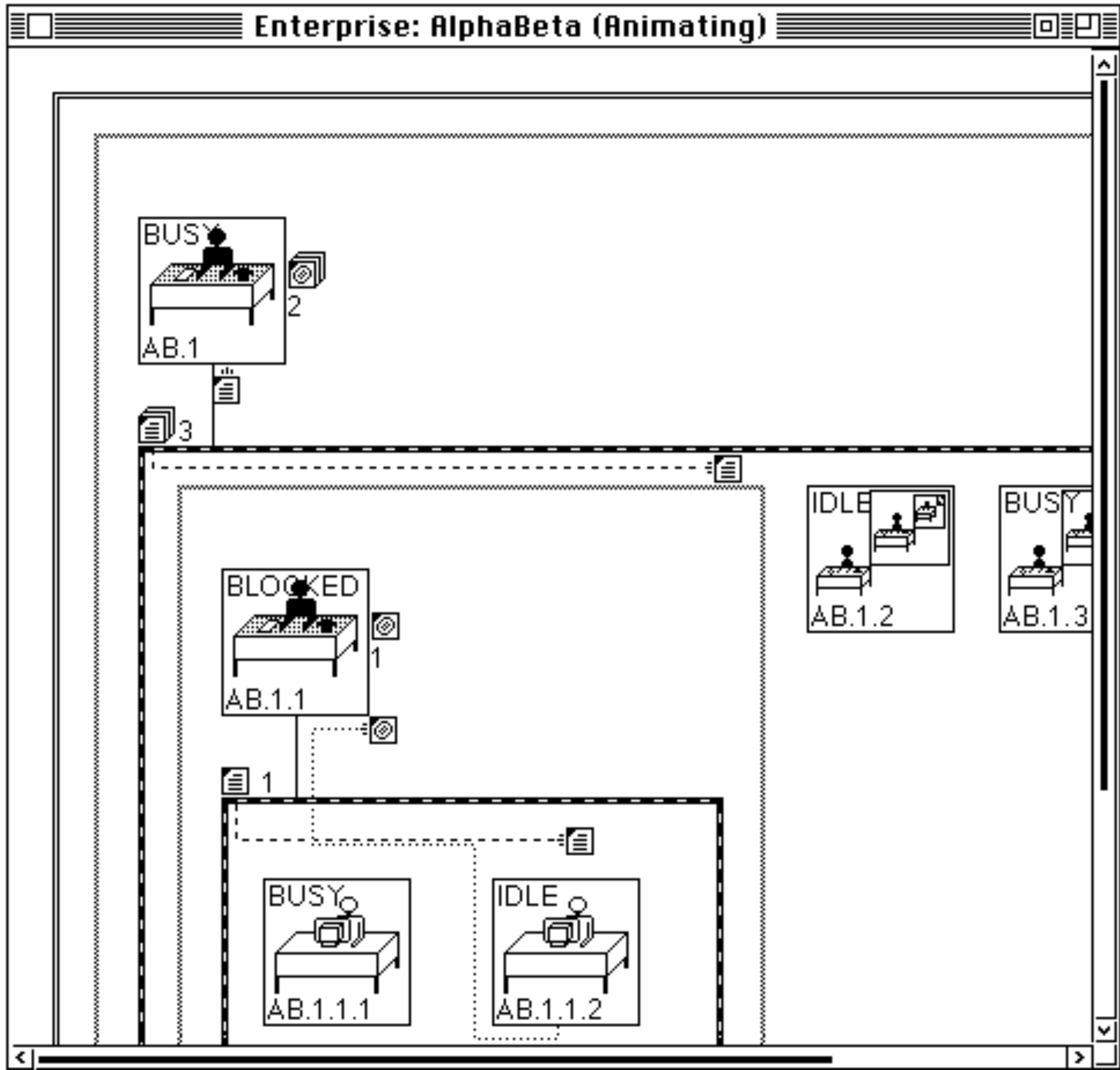

Figure 7: The *Animation* view of the *AlphaBeta* program.

This example uses *division* assets in a recursive divide-and-conquer application, but the number of processes and the size of the message queues have been reduced for brevity. The *enterprise* contains a *division* with a *receptionist* (AB.1) and subordinate *division* that has a replication factor of three (AB.1.1, AB.1.2 and AB.1.3). Each subordinate *division* contains a *receptionist* with a replicated *representative*. Note that for most assets, replicas are numbered left to right. However, the replicas in *division* assets are structured hierarchically instead of linearly.

Collapsing and expanding assets in the *Animation* view provides a clustering mechanism [Tay92]. Clustering is useful during debugging since it reduces the clutter caused by displaying too much inappropriate detail and allows the user to focus on the important relationships. Two of the subordinate *divisions* (AB.1.2 and AB.1.3) have been collapsed, but the other (AB.1.1) is expanded.

The *Animation* view displays two message queues. Incoming messages are in the *input queue* above the asset, and replies to previously sent messages are in the *reply queue* to the right of the asset. These locations correspond to the logical structure of the user's code where calls are received at the start and replies are received in the body. Replicated assets share a common input queue that is displayed above and to the left of the replicated assets. However, each replica has its own reply queue. Messages are represented by icons that move along the paths between assets and into the message queues.

A message queue displays the number of messages it contains. A message queue icon shows zero (no visible icon), one (a single message icon) or many (a message icon with two others behind it) messages. The number beside the queue icon indicates the exact count. When a message arrives at a queue this count is incremented and when a message is removed from the queue to be processed by an asset, the count is decremented.

When the animation is active but stopped, the message queue menu can be used to select any message it contains and to display its sender, parameter values and any other information that is placed in the message event by the event logging process.

In Figure 7, each asset is either *busy* (processing a task), *idle* (waiting for a message to invoke a task) or *blocked* (waiting for a specific reply). Asset AB.1 has just sent a message to its replicated subordinate *division*. The message appears below AB.1 and will move to the input queue of the replicated *division* as the animation proceeds. A message icon looks like a memo with four lines and a bent upper left corner. Currently, the replicated *division's* input queue contains three messages. Since asset AB.1.2 is idle, a message is moving from the input queue to it. Similarly, a message is moving from the input queue of the replicated representatives (AB.1.1.1 and AB.1.1.2) to the idle asset, AB.1.1.2.

Representative AB.1.1.2 has completed a task and replied to its caller, AB.1.1. The reply message is shown on its way to the reply queue of AB.1.1.2. A reply icon looks like a memo that has been stamped as received. Note that the message path of a reply begins at the bottom of the replying asset, corresponding to the structure of an asset's code where the return statement is usually at the end.

4.2 States

At run-time, *Enterprise* assets become processes. A process communicates with other processes by sending messages. As an asset executes, it can be in one of four states: idle (waiting to receive a message), busy (executing), blocked (waiting for a reply), and dead. An asset changes state in response to events that affect it.

The state of a collapsed asset is determined by the states of its components. If at least one component is busy, the asset is busy. If no component is busy and at least one is blocked, the asset is blocked. If no component is busy or blocked and at least one is idle, the asset is idle. Otherwise all of the components must be dead, so the asset is dead.

The state of an asset is indicated in the *Animation* view by one of two (user-selectable) mechanisms: color or state name display. Icons for busy assets are green, icons for idle assets are yellow, icons for blocked assets are red and icons for dead assets are black.

4.3 Events

Assets change state in response to events that occur when the program is running. The event logging process monitors programs as they run, identifies when important events occur, and writes event records to an event file, maintaining the original partial ordering between the events. The animation system reads the events from the event file and updates the display. Seven events are supported: SentMsg, RcvdMsg, Block, SentReply, RcvdReply, DoneMsg and Die. Figure 8 is a state-transition diagram that shows the relationship between the asset states (represented by circles) and the events (represented by arrows).

The event file is an ASCII text file. Each event starts on a new line that begins with a # character and a space followed by an event type and parameters separated by spaces. An optional information string can follow on the next line. The information string is displayed by the user-interface when the user inspects message contents.

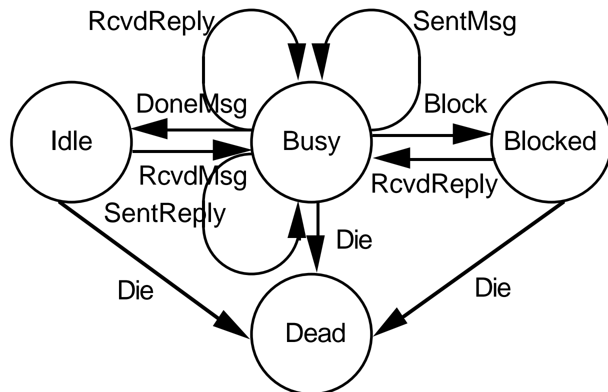

Figure 8: The asset state transition diagram.

Event parameters depend on event types. They include asset names, message tags and integers representing times. Asset names are the names from the graph file with replica numbers appended to them. Tags are integers that are used to associate SentMsg events with RcvdMsg events and SentReply events with RcvdReply events. Times are measured from some arbitrary start time in milliseconds and refer to the time that the event was inserted into the event file. The sequence of times must be monotonically non-decreasing.

SentMsg

When the event logging process detects that an asset has sent a message to another asset, it inserts a SentMsg event in the event file. The information string contains the names and values of all message parameters. During animation, a message moves from the sender to the input queue of the receiver where the message count is incremented. The sender must be busy and it does not change state. The receiver does not change state.

RcvdMsg

When the event logging process detects that an asset has received a message and started processing the task that the message invokes, it inserts a RcvdMsg event in the event file. During animation, the receiver decrements its input queue counter. The receiver then changes its state from idle to busy.

DoneMsg

When the event logging process detects that an asset has finished executing a message, it inserts a DoneMsg event in the event file. During animation, the receiver changes its state from busy to idle.

SentReply

When the event logging process detects that an asset has has sent a reply message to its caller, it inserts a SentReply event into the event file. The information string contains the names and values of all message parameters. During animation, a message moves from the sender to the reply queue of the receiver and the message count is incremented. The sender asset must be in the busy state but the receiver may either be busy or blocked.

RcvdReply

When the event logging process detects that an asset has accessed a message reply, it inserts a RcvdReply event into the event file. During animation, the message count in the reply queue is decremented. The asset that receives a reply may either be busy or blocked. If the asset was blocked with the same tag as the RcvdReply it becomes busy.

Block

When the event logging process detects that an asset has tried to access a result computed by another asset, and the result is not available, it inserts a Block event into the event file. The Block event includes a tag that indicates the reply it is waiting for. During animation, the asset state changes from Busy to Blocked.

Die

If the event logging process determines that an asset is not responding for some reason, it inserts a Die event into the event file. During animation, the asset becomes dead, but the message queues are not affected so that the user can examine them, after the event. The asset can be any state before this event.

4.4 The Animation Architecture

The object-oriented animation architecture described in this section is new and application independent. It has two main components, one is asynchronous and the other is synchronous. The asynchronous component has two responsibilities. It must process the events at the correct animation time. However, since we want the user to be able to interact with the system during animation, it is also responsible

for user events as well. The synchronous component of the animation system is responsible for animating messages.

The Asynchronous Component

Several new classes were added to the user-interface to support animation and several behaviors were added to the existing classes. When the *Animation* view is displayed, the asset graph is modified. Each replicated asset is wrapped in an instance of *ReplicatedAsset* that contains the original asset together with a list of replicas that are constructed by copying the original asset. The copies are identical, except that each is given a different id number. As an animation proceeds, the states of these replicas may diverge. The ReplicatedAsset is responsible for drawing the connections between replicas, much like *ExpandableAssets* do for their components.

Two new responsibilities are added in the *Asset* class, knowing the input message queue and knowing the reply message queue. Both queues are instances of the subclasses of *MessageQueue*, *InputQueue* and *ReplyQueue*. A *MessageQueue* contains an ordered collection of messages, which are instances of class *Message*. The display method in *Asset* checks to see if animation is active and if so, allocates room for the message queues when it computes its bounding rectangle. When an asset is told to draw itself, it also tells its message queues to draw themselves.

Message queue selection is implemented by augmenting the message that is sent to an asset to ask it for its sub-asset that contains the cursor point. An asset now considers its two queues as candidates in addition to its component assets. A *MessageQueue* determines if it contains the cursor point by testing if the point is within its screen extent.

An instance of class *EventQueue* is responsible for knowing the start time for an animation and the events from an event file. It is created when the *Animation* view is displayed. That is, to speed up event processing, the event file is parsed and all events are created before the animation begins. The animation start time is set when the user actually starts an animation.

When the event file is parsed and instances of class *AnimationEvent* are created, each event time is translated to a time relative to the start time for its event queue. When the animation is active, the control loop for the window sends a message to the program every time through the loop. The program responds by telling the animation event queue to process its animation events. The event queue processes its events in order until the event time plus the start time catches up to the current time. Control is then returned to the control loop which checks for user input. In this way the animation system only takes control periodically and, when it does, only for a short time. This allows users to interact with the system during an animation. For example, the user could pause the animation.

Each animation event represents one event from the event file. In addition to the event time, an animation event contains a collection of animation messages. Each of the animation messages consists of a receiver asset, a message selector, and an array of arguments for the message. One event may translate into several animation messages. For example, a SentReply event translates to two animation messages: one to tell the sending asset it has sent a reply and one to tell the receiving asset it has been sent a reply. The set of messages for one event is treated as a transaction; if one message is sent they all are. There is a subclass of the abstract superclass, *AnimationEvent*, for each type of event. Each event sub-class need only implement creation messages. All other messages are implemented in *AnimationEvent*. In addition, the asset classes implement methods for each animation message sent by an animation event. The responsibilities include changing state, updating message queues, and modifying the display.

Assets have input and reply message queues. Each queue contains an ordered collection of messages. They are displayed either above or beside an asset. Messages move along the paths between assets and into the queues in response to SentMsg and SentReply events. For a SentMsg event, a message moves from below the sending asset to just above the receiving asset and then into its input queue. For a SentReply event, a message moves from below the replying asset to just below the receiving asset and into its reply queue. Although messages must move different distances on the display screen, these

distances are not necessarily indicative of the actual communication distances. Therefore a message moves from one asset to another in (user adjustable) constant time. For example, with replicated assets, the replicas will be different distances from the calling asset due to the way that *Enterprise* displays assets hierarchically. To compensate, messages with longer screen travel distances move faster to maintain a constant time interval.

Because the destination queue is part of the receiver, animating the message is actually done by the receiver. When a SentMsg or a SentReply event occurs, the receiver is informed. The receiver creates a message, inserts it into its message queue and marks it as pending, determines the path it must follow to move from the sender into its queue, and asks the message to animate itself. When the message reaches the message queue, the receiver removes the pending mark and increments the counter for its message queue. The user can examine any message in a message queue even if it is pending (the animation has not yet shown it reaching the queue).

A message is received when the receiver gets a RcvdMsg or a RcvdReply event for the message. When this occurs, the receiving asset will remove the message from its message queue. If the message is marked as pending, the receiver will remove the message from the animation queue so it disappears at the next animation step. If the message is not pending then the receiver decrements its message queue counter.

<u>The Synchronous Component</u>

Animation of messages and busy assets are done synchronously. The program maintains an instance of *AnimationQueue* that holds objects to be animated. When the program tells its event queue to process events, it also tells its animation queue to animate its objects. The animation queue checks to see if it is time to perform the next step of the animation and, if it is, sends an animate message to every object in its queue. If it isn't time, the queue does nothing. The time between steps is a constant. The class of each object in the queue must support the animate message to perform one step of the animation.

A message in the animation queue animates itself by moving along a pre-computed path in steps. The path was computed by the asset that created the message. This asset computed the location of the sender and receiver and computed a set of points along the path between them. The path was stored in the message before the message was added to the animation queue. Whenever a message receives an animate message, the message moves itself to the next point on its path, then deletes the point from its path. If a message reaches the end of its path, it removes itself from the animation queue, tells the receiver to mark it as not pending and tells the receiver to increment its message queue counter.

5. Conclusions

This paper described the object-oriented component of the *Enterprise* programming environment for developing distributed applications that execute concurrently on a network of workstations. The object-oriented components provide a new anthropomorphic model for parallel computation. The simplicity of this model:

1. makes it easier to learn than other models of parallel computation,
2. has allowed programmers to write parallel programs more quickly than with other models and
3. has reduced the complexity of the user-interface and the other *Enterprise* components so they could be designed and implemented quickly.

Enterprise includes an animation component that:

1. has a new architecture that supports asynchronous and synchronous events,
2. is a valuable tool for understanding the complexity of parallel computations and
3. is independent of *Enterprise* so that it can used for other applications.

Our experience with the object-oriented components of *Enterprise* have also provided some insights into the use of object-oriented computing in general and ST-80 in particular.

1. The advantages obtained by using the extensive user-interface libraries of ST-80 outweigh the perceived disadvantages. The

efforts required to combine object-oriented user-interface code with traditional C code were minimal. The execution time performance problems of ST-80 are insignificant in user-interfaces, even though in this application the user-interface is fairly CPU intensive during animation.

2. Although Smalltalk has not been used extensively to construct user-interfaces where object motion is an important factor, the *Enterprise* experience illustrates its power for such applications.
3. The lack of support for multiple inheritance is a significant problem in Smalltalk when the application depends on a real-world analogy.

The success of the *Enterprise* project is largely due to its object-oriented components. In fact, several members of the research group who had severe doubts about the utility of the object-oriented approach are now firmly committed to the use of object-oriented technology for user-interfaces in particular and for embedded applications in general.

Acknowledgements

The *Enterprise* project has benefitted from the efforts of many people, including: Paul Iglinski, Paul Lu, Ron Meleshko, Ian Parsons, Carol Smith and Zhonghua Yang. This research was supported in part by research grants from the Central Research Fund, University of Alberta, the Natural Sciences and Engineering Research Council of Canada, grants OGP-8173 and 107880 and a grant from IBM Canada.

References

[Lam78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *CACM*, Vol. 21, No. 7, pp. 558-565, 1978.

[LMP92] G. Lobe, P. Lu, S. Melax, I. Parsons, J. Schaeffer, C. Smith and D. Szafron. The Enterprise Model for Developing Distributed Applications. Technical Report TR 92-20, Dept. of Computing Science, University of Alberta, 1992.

[LP91] W. LaLonde and J. Pugh. *Inside Smalltalk Volume II*. Prentice-Hall, Englewood Cliffs N.J., 1991.

[LSW86] D. Lanovaz, D. Szafron and B. Wilkerson. The Synergism of Logic-Based Programming and Software Engineering: A Programming Environment Approach. *CIPS Edmonton '87 Intelligence Integration Conference Proceedings*, pp. 43-53, November, 1987.

[LVC89] M.A. Linton, J.M. Vlissides and P.R. Calder. Composing User Interfaces with InterViews. *IEEE Computer,* Vol. 22, No. 2, pp. 8-22, 1989.

[Par93] I. Parsons. An Appraisal of the Enterprise Model. M.Sc. thesis, Dept. of Computing Science, University of Alberta, 1992.

[SSG91] A. Singh, J. Schaeffer and M. Green. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Transactions on Parallel and Distributed Systems,* Vol. 2, No. 1, pp. 52-67, 1991.

[Tay92] D. Taylor. A Prototype Debugger for Hermes. *Cascon '92*, IBM Canada Ltd, Toronto, pp. 29 - 42, November, 1992.

[WWW90] R. Wirfs-Brock, B. Wilkerson and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[You92] D. Young. *Object-Oriented Programming with C++ and OSF/Motif*. Prentice-Hall, Englewood Cliffs N.J., 1992.