University of Alberta

Program Design and Animation in the Enterprise Parallel Programming Environment

by

Greg Lobe
Duane Szafron
Jonathan Schaeffer

Technical Report TR 93-04
March 1993

Program Design and Animation in 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 implements 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.

Keywords: Object-oriented, Smalltalk, programming environment, user-interface, animation, distributed computing

1. Introduction

This paper describes how object-oriented techniques were used to design and implement components of the *Enterprise* programming environment. *Enterprise* supports the development of distributed applications, written in the C programming language, that run on a network 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 for 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. With multiple processes running 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 components. Most 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 by using an analogy based on the parallelism in a business organization. Most parallel computations can be structured hierarchically, with "higher-level" management processes performing executive functions, and "lower-level" subordinate processes carrying out designated tasks. Since business enterprises efficiently coordinate many asynchronous individuals and groups, the analogy is beneficial to designing, 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 is to be executed 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 user's 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 process that executes the asset `func`. The caller continues executing and only when it accesses the result of the function call `(a = result)` does it block and wait for the result. *Enterprise* also supports passing parameters by reference.

Enterprise consists of 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 sends across a network by the pre-compiler. The run-time executive controls program execution (process/processor assignment, establishing communication links, monitoring the network load). More information about *Enterprise* including the anthropomorphic parallel programming model, the system implementation and a user appraisal can be found in [LMP92], [SSW92] 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 features.

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

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

2. Designing Programs Using Enterprise

This section presents a simple example of how *Enterprise* can be used to construct a distributed program. Consider a program (called Simulation) that displays a group of fish swimming across a display screen. This problem was contributed by a research group in our Department and is obviously more complex than portrayed by the following description. There are three fundamental operations in the program (*Model*, *PolyConv* and *Split*) with the following functionality and pseudo-code.

- The main procedure, *Model*, computes the location and motion of each object in a simulation frame, stores the results in a file, calls *PolyConv* to process the frame and goes to the next frame.

```
Model()
{
   for each frame
      {
         /* compute location and motion of objects */
         PolyConv( frame );
      }
}
```

- *PolyConv* reads a simulation frame from the disk file and performs some data format transformations, viewing transformations, projections, sorts and back-face removal. It then calls *Split*, passing it a transformed frame and a sequence number.

```
PolyConv( frame )
{
   /* perform transformations and projections */
   Split( frame, polygons );
}
```

- *Split* performs hidden surface removal and anti-aliasing and then stores the rendered image in a file.

```
Split( frame, polygons )
{
   /* hidden surface removal and anti-aliasing */
}
```

Examining the structure of the program shows that *Model* consists of a loop that, for each frame in the simulation, performs some work on the frame and calls *PolyConv* with the results. *PolyConv* manipulates the image received from *Model* and calls *Split*. *Split* does the final polishing of the frame and writes the final image to disk.

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

When *Enterprise* is started, the *Enterprise* window contains a single view called the *Enterprise View*. It contains the icon for a single *enterprise* asset that represents the new program. Associated with each asset is a context sensitive pop-up menu. For example, if the user selects *Name* from the asset menu of the *enterprise* and types the word *Simulation* into the dialog box that appears, the *enterprise* would be named *Simulation* and appear as in Figure 1. Note that the *Enterprise* user-interface is implemented in ST-80 which uses the host windowing system. The

figures in this report were generated on the Macintosh implementation of the *Enterprise* user-interface and look slightly different in X windows [GKM90] or Sun OpenWindows [Sun91].

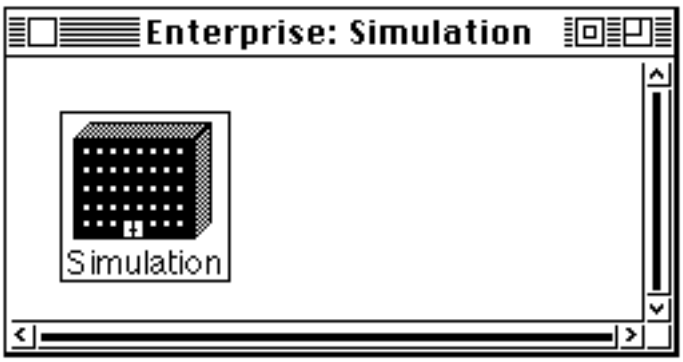

Figure 1: A new program called Simulation.

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 single *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 in the *line*.

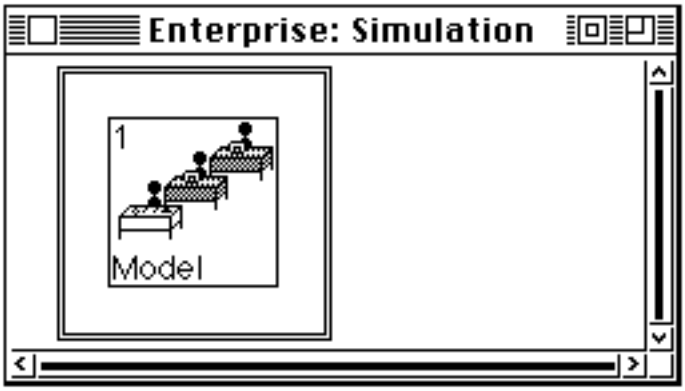

Figure 2: A sequential program that contains a *line* called *Model.*

If the user selects *Expand* from the asset menu of *Model*, it is expanded to reveal its two components. Since we want three components, the user selects *AddAfter* from the last

component's menu to add a third asset and then names the new assets, *PolyConv* and *Split*. The user can then select *Code* from the menu of each asset in turn and enter the C source code into the text editor window that appears, as shown in Figure 3.

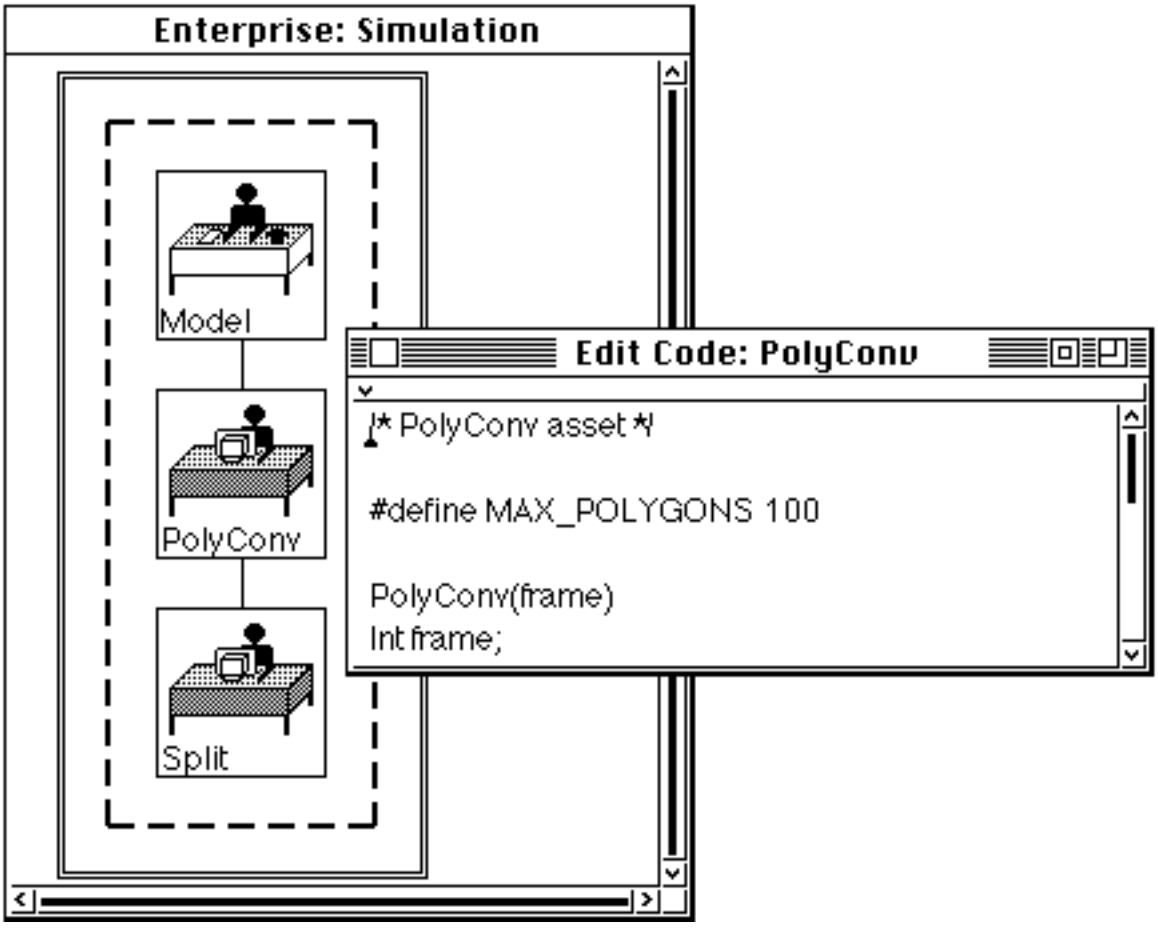

Figure 3: Editing the C source code for *PolyConv*.

The double-line rectangle represents the *enterprise*. The dashed-line rectangle represents the *line* and each 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*.

If the user selects *Compile* from the *Enterprise* view menu, the *Enterprise* pre-compiler inserts the parallelization code, compiles the program and reports any errors in a window. The user can then select *Execute* from the *Enterprise* view menu and *Enterprise* finds as many processors as are necessary to start the program, initiates the processes and monitors the load on the machines.

One of the strengths of the *Enterprise* model is that it is easy to experiment with alternate parallelization techniques without changing the 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 multiple 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.

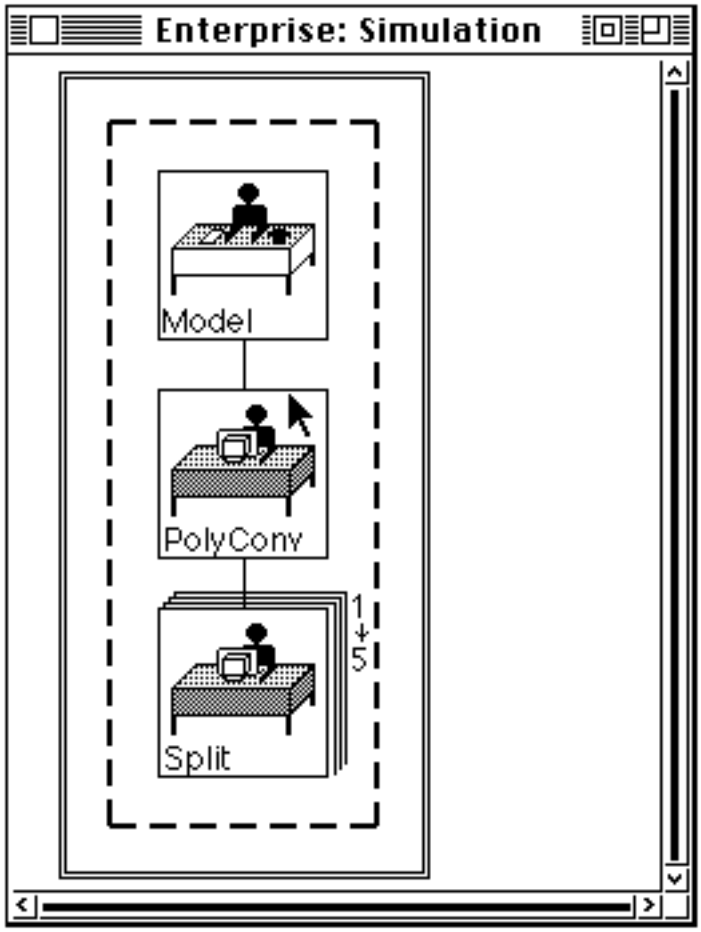

Figure 4: A replicated asset.

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 is supported by 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. Under Unix, 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

---

™ Unix is a trademark of Bell Laboratories.

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 PolyConv*, 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 *PolyConv*.

If an asset is selected, a context sensitive menu is displayed. The menu contains only those operations that are currently 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. In this way, it is 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*.

Even when it is assumed that assets are responsible for knowing their locations, there are several approaches for determining the selected asset and transferring control to it so that its menu can be displayed and the appropriate action taken.

The following naive approach was tried first. The ST-80 implementation of MVC provides a default behavior that passes control to the innermost view that contains the cursor. This is implemented by maintaining a list of scheduled controllers (the controller of each window). Each of the scheduled controllers is sent a message in a polling loop. If a controller's view has the cursor, it takes control and asks its view if one of its sub-views wants control. If one does, the controller gives control to the sub-view's controller, otherwise it keeps control itself. The sub-view's controller behaves the same way. Thus, the controller for the innermost sub-view that contains the cursor gets control.

Since assets are views, the method that determines if a sub-view wants control was re-implemented. This was necessary since ST-80 assumes that sub-views are always displayed. If an asset is collapsed or has no components, the method returns the asset itself. Otherwise the method invokes the original method that recursively finds a component of the asset that wants control. Once a controller has control and knows that none of its sub-views wants control, it displays its menu and it processes the user's choice.

Unfortunately, this approach failed. There were times when the wrong menus would be displayed. Clicking on an asset would bring up the menu for one of its components, its parent, or even one of its parent's parents. Clicking at the same location again would sometimes display the same menu, but would sometimes display the right one or a completely different one. It seemed

like the wrong controller was taking control. This behavior was caused by two different phenomena. First, each asset asked the cursor for its location. When the cursor was moved between the times that two assets queried it, each would receive a different point. Second, the control method was not actually as simple as described previously. The method in the controller that asks sub-controllers if they want control is sent from a loop. The loop iterates until the asset no longer has the cursor. When a menu was displayed, the active control loop was initiated from a controller in a loop that was initiated from a controller in a loop, etc. When the active controller finished processing the user's choice, that loop would not end. The next time the user clicked the mouse the controller would assume that it had the cursor (because it was active), would find that no sub-view wanted control, and would display its own menu.

Our second approach alleviated this problem. The *Enterprise* view determines the coordinates of the cursor and asks the *enterprise* which asset should be selected. The *enterprise* either returns the selected asset or the *UndefinedObject*, *nil*, if no asset contains the cursor point. In the former case, the selected asset is given control to display its menu and perform the selected action. In the latter case, the *Enterprise* view displays its own menu and performs the selected action.

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. First the asset draws its icon. Then it displays its name in the lower left corner of the icon. If the asset is replicated, the replication is indicated by drawing lines above and to the right of the icon to simulate a stack of icons, and by displaying the number of replications 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 appropriate connections for the various *Asset* sub-classes.

3.3 Communicating with the Other Enterprise Components

Although the user-interface is implemented in ST-80, the other two *Enterprise* system 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 *Enterprise* components.

Graph, Event and Preference Files

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 their descriptions in the same format. Appendix A contains a description of the *Enterprise* graph file format.

Graph files are created and edited by the user-interface. When the user selects the *Save*, *Compile*, or *Run* commands from the *Enterprise* view menu, the *enterprise* is asked to store a representation of itself in a graph file whose name is the *enterprise* name with a ".e" 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 to represent the saved program.

The pre-compiler uses the information contained in 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. Appendix B contains a description of the *Enterprise* event file format.

Enterprise maintains a preferences file. When the user-interface first starts, it looks in the current directory for a file named *Enterprise.prefs*. 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 Managing Source Code

When a new program is created, *Enterprise* creates a new sub-directory of the current directory with the same name as the program. It then creates other sub-directories of this new directory to organize the files used by the program. The following sub-directories are created:

Assets — This directory holds the C source code for all of the assets. Each asset's code is stored in a file ending with a ".e". The pre-compiler parses these files and produces corresponding files ending with ".c". For example, the user's code for an asset named *Model* will be stored in *Model.e*. The pre-compiler produces the corresponding file *Model.c*.

Source — This directory holds C source code for internal procedures used by assets.

Include — This directory holds header files for all code in the Assets and Source directories. The *Enterprise* pre-compiler and the C compiler search this directory when processing #include directives.

Data — If the user specifies input and output files from the run parameters dialog box to re-direct program input or output, they will be stored here.

Obj — This directory holds sub-directories that contain the object ".o" files for each machine type on the network. This feature is necessary to support the execution of applications on a network of heterogeneous computers.

Bin — All executable programs produced by *Enterprise* will be stored here. A separate executable program is required for each type of computer on the network.

Sys — This directory holds all of the system generated data files for the program, such as the graph and event files.

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, when the user-interface is running on 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 windows 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 event polling loop in the controller for the *Enterprise* view 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.

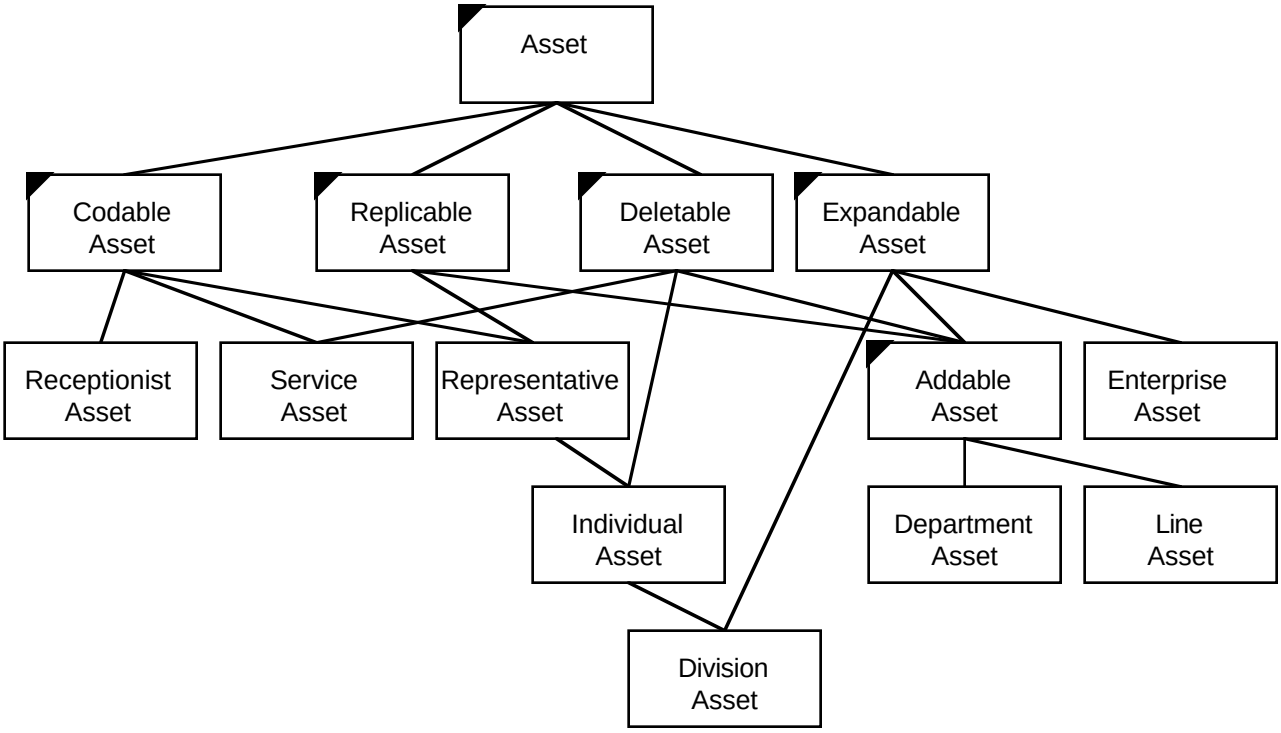

Figure 5: The asset inheritance graph.

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 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 code. An *EnterpriseAsset* is expandable, has no 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.

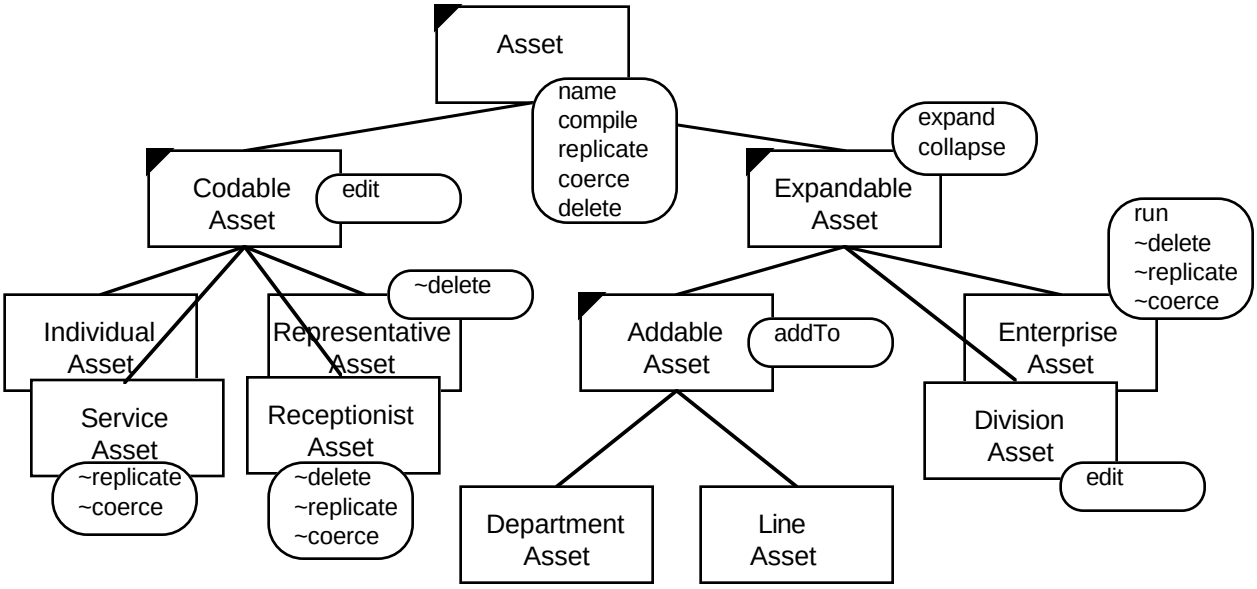

Figure 6: The asset inheritance tree.

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 code editing methods were then re-implemented in *DivisionAsset*. In addition to these changes, the *Asset* class 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.

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 produced by an external Unix event monitoring process that receives messages from the run-time executive. The interface assumes that the events are partially ordered [Lam78] by the monitoring process. 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 the real time program execution. The proportionality factor can be adjusted by the user to speed up or slow down the animation. The user can also execute 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. The *Animation* view of the Simulation program is shown in Figure 7 and the *Animation* view of a recursive *AlphaBeta* search program is shown in Figure 8.

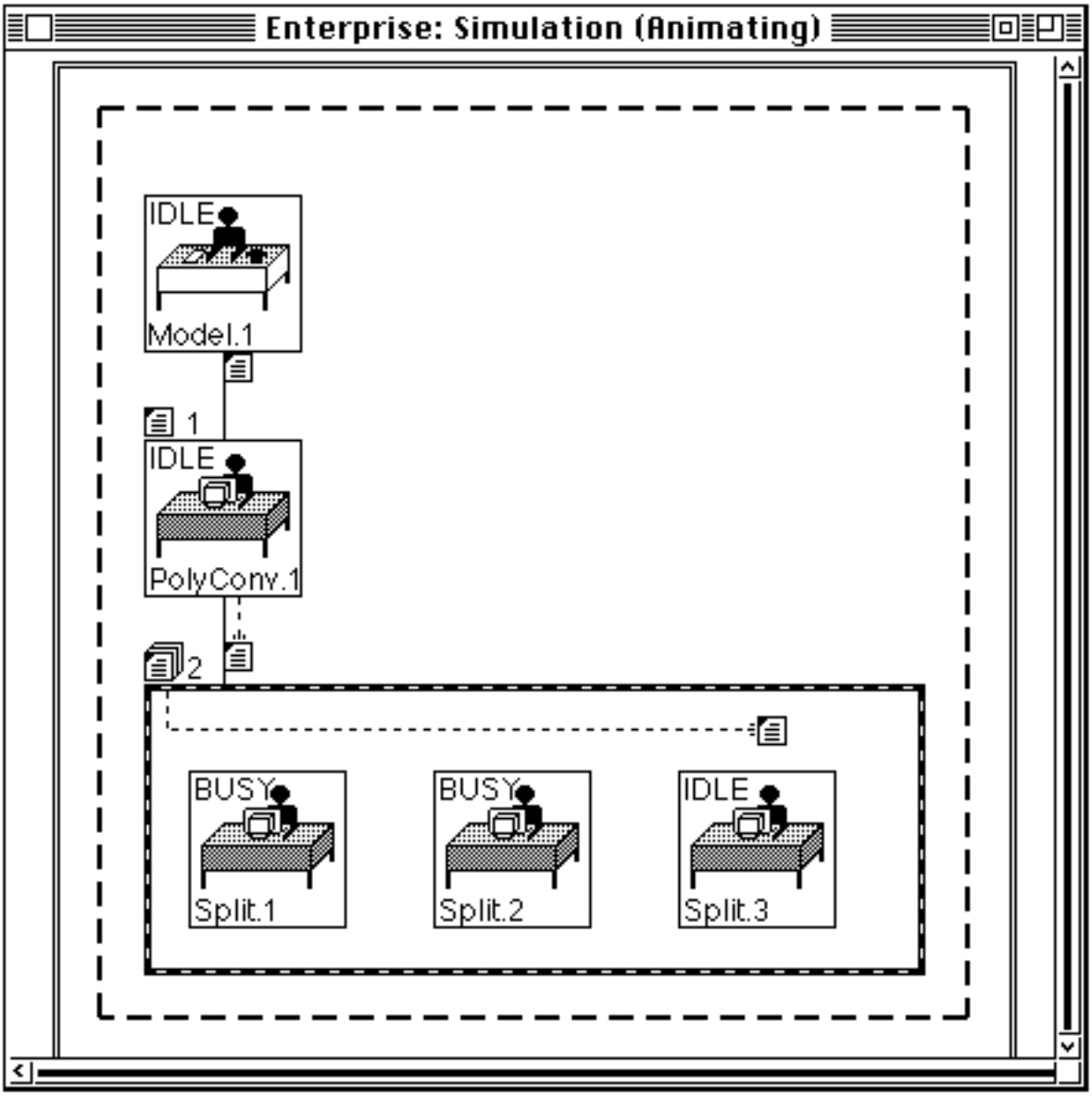

Figure 7: The *Animation* view of the Simulation program.

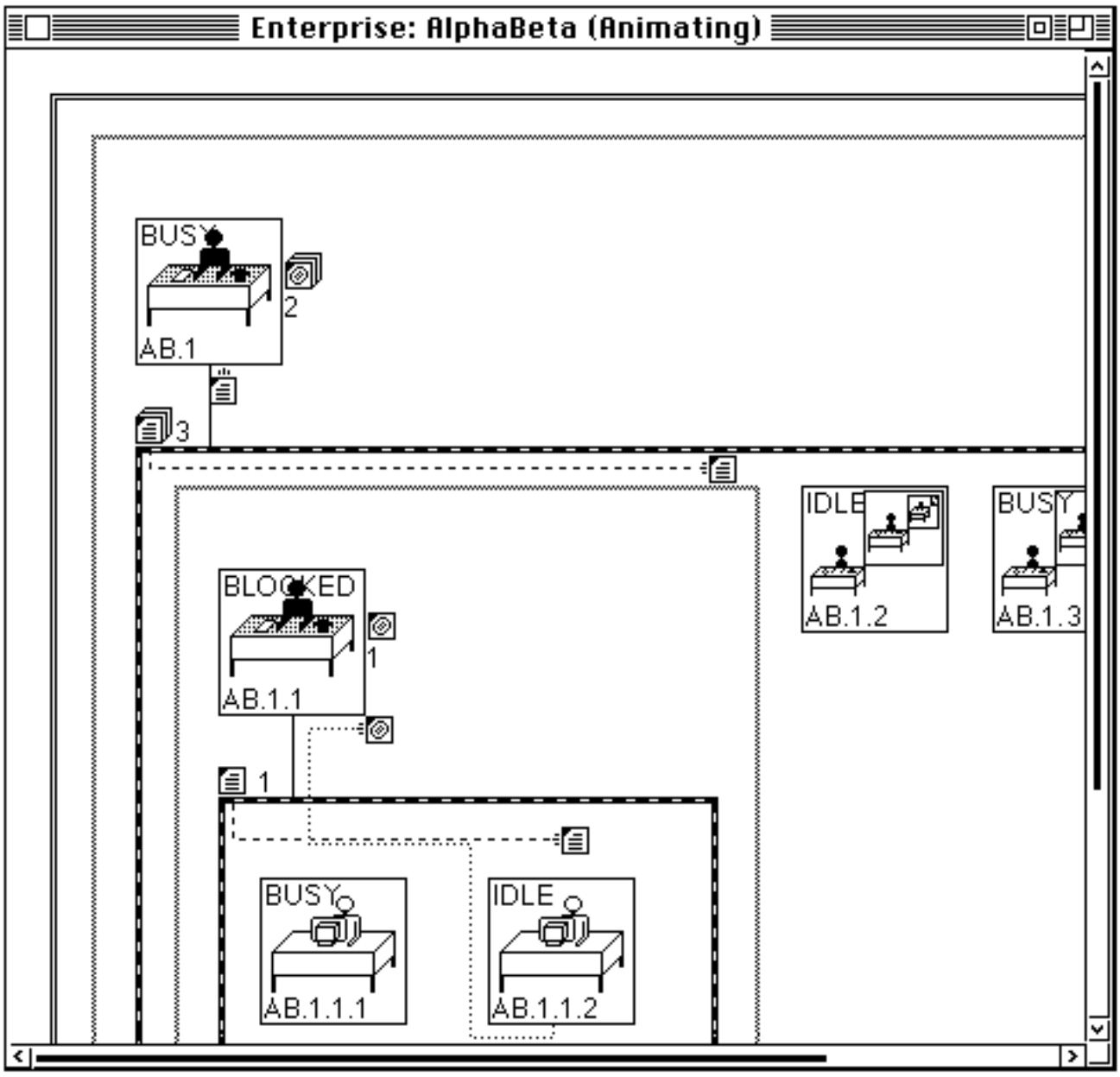

Figure 8: The *Animation* view of the AlphaBeta program.

Each replica from a replicated asset is displayed as a separate icon, 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 itself has choices for starting the animation from the beginning, pausing or resuming the animation, single stepping through events, setting the speed of the animation and replacing the *Animation* view by the *Enterprise* view.

Assets can be collapsed and expanded in the *Animation* view to provide a clustering mechanism [Tay92]. Clustering is a useful abstraction technique during debugging since it reduces the clutter caused by displaying too much inappropriate detail and allows the user to focus on the important relationships. For example, Figure 8 shows a *division* (enclosed by a wavy-line rectangle) that contains a *receptionist* (AB.1) and a *division* with a replication factor of three. The left *division* is expanded to show its component assets while the two *division* assets on the right are collapsed. Each of these *divisions* contains a *receptionist* and a *representative* with replication factor of two. A *representative* is a leaf node of a division hierarchy.

The *Animation* view displays two message queues. Incoming messages are queued in the *input queue* above the asset, and replies to previously sent messages are queued 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 of the code and replies are received in the body of the code. 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. When a message arrives at a queue this number is incremented and when a message is removed from the queue to be processed by an asset, the number 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.

Replicas are named by appending an id number to the base asset name assigned by the user. The id numbers for each asset are generated in sequential order starting at 1. Replicas are numbered left to right as shown in Figure 7. However, the replicas in *division* assets are structured hierarchically instead of linearly as shown in Figure 8.

Figure 7 shows an animation of the simulation program at a specific point in time. Each asset is either *busy* (processing a task) or *idle* (waiting for a message to invoke a task). *Model* has just sent its last message to *PolyConv* and has become idle. The message appears below *Model* and will move to *PolyConv*'s input queue as the animation proceeds. Currently, *PolyConv*'s input queue contains one message. However, *PolyConv* has just sent a message to *Split*, completed its previous invocation and is now idle. Therefore, the message in its input queue will be received and removed from the queue, momentarily. At this point, *PolyConv* will change its state to busy. The message that *PolyConv* sent to *Split* will move left into the common input queue for the replicated asset and increment the queue count to 3. Note that message queue icons show 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. Two of the replicated *Split*

assets, *Split.1* and *Split.2*, are currently busy. However, *Split.3* has completed its task and is currently idle. Since there are messages in the input queue waiting to be processed, a message is moving from the queue to *Split.3*.

Figure 8 shows an animation of an *AlphaBeta* tree search program [MRS87] that illustrates message replies. Note that in this example, the number of processes and the size of the message queues have been reduced for brevity. This application was created using *division* assets that allows one to easily write parallel recursive divide-and-conquer applications. The application consists of a *division* that contains a *receptionist* (AB.1) whose subordinate *division* 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*. Two of these subordinate *divisions* (AB.1.2 and AB.1.3) have been collapsed, but the other (AB.1.1) is expanded.

At the moment represented in Figure 8, 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. 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, busy, blocked, and dead. An asset changes state in response to events that affect it.

Idle — An idle asset is one that is not currently executing. It is waiting to receive a message. The next message sent to it will be received immediately.

Busy — A busy asset is one that is executing code in response to a message from a caller asset. It can send messages to other assets and receive replies from them, but cannot receive a message from another caller until it completes the active message. All messages sent to it are put in its message queue.

Blocked — A blocked asset is one that has stopped execution to wait for a reply to a message it has sent. This occurs when an asset tries to access the return value from an asset call that has not yet replied. All messages sent to it are put in its message queue.

Dead — A dead asset is one that has stopped execution because of some kind of error. The *Enterprise* executive has determined that it can no longer communicate with any other asset. The asset cannot send messages and ignores any messages sent to 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 9 is a state-transition diagram that shows the relationship between the asset states (represented by circles) and the events (represented by arrows).

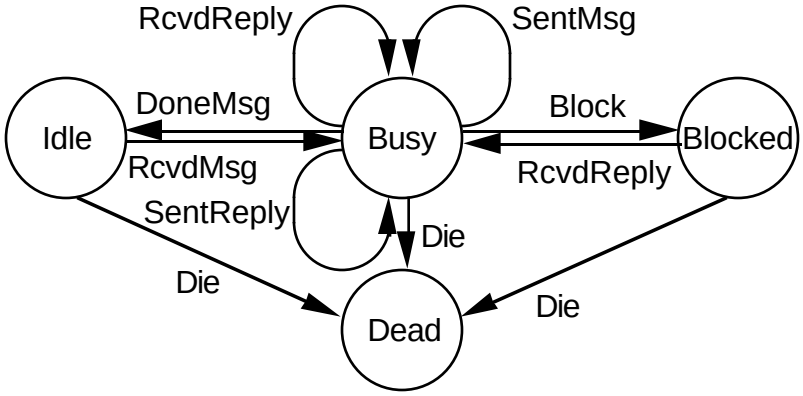

Figure 9: The state transition diagram for *Enterprise* assets.

The event file is an ASCII text file. Each event starts on a new line. It begins with the # character and a space followed by an event type and its parameters separated by spaces and ends with a new line character. 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.

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 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 of the Animation Architecture

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 a RcvdMsg or a RcvdReply event occurs. The receiving asset removes the message from its message queue. If the message is marked as pending, the receiver also removes 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.

The Synchronous Component of the Animation Architecture

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 describes the object-oriented component of the *Enterprise* programming environment for developing distributed applications that execute concurrently on a network of workstations. These 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 from 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

[GKM90] J. Gettys, P. Karlton and S. McGregor. *The X Window System, Version II*. Software - Practice and Experience, Vol. 20, No. 2, pp. 35-67, 1990.

[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.

[MRS87] T.A. Marsland, A. Reinefeld and J. Schaeffer. Low Overhead Alternatives to SSS*. *Artificial Intelligence,* Vol. 31, No. 1, pp. 185-199, 1987.

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

[Sun91] Sun Microsystems, Inc. *OpenWindows DeskSet Reference Guide*. Sun Microsystems Inc., 1991.

[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.

[SSW92] D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu and C. Smith. The Enterprise Distributed Programming Model. *Programming Environments for Parallel Computing*, N. Topham, R. Ibbett and T. Bemmerl, editors, Elsevier Science Publishers, pp. 67-76, 1992.

[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.

Appendix A: The Enterprise Graph File Format

This appendix describes the format of the *Enterprise* graph file using extended BNF notation.

Notation

```
<abcd>*  means 0 or more occurrences of <abcd>
<abcd>+  means 1 or more occurrences of <abcd>
```

Syntax

```
<graph>        ::= <asset>
                   <service>*

<asset>        ::= <name> <simpType> <min> <max> <order> <debug> <opt>
                   <options>
               |   <name> <compType> <min> <max> <order> <debug> <opt>
                        <childcount>
                   <options>
                   <asset>+

<service>      ::= <name> service <debug> <opt>
                   <options>

<name>         ::= <string>

<min>          ::= <positive integer>

<max>          ::= <non-negative integer>

<order>        ::= ORDERED | UNORDERED

<debug>        ::= DEBUG | NDEBUG

<opt>          ::= OPTIMIZE | NOPTIMIZE

<simpType>     ::= individual | representative

<compType>     ::= line | department | division

<childcount>   ::= <positive integer>

<options>      ::= CFLAGS <flags>
                   EXTERNAL <libraryList>
                   INCLUDE <machineList>
                   EXCLUDE <machineList>

<libraryList>  ::= <string>

<machineList>  ::= <string>

<flags>        ::= <string>
```

Semantics

`<graph>`

A graph represents the entire *Enterprise* program. It consists of an asset definition followed by 0 or more service definitions. The file can be parsed from top to bottom to perform a depth-first traversal of the graph.

`<asset>`

An asset can either be simple or composite. Simple assets are either *individuals* or *representatives*. Each is represented by one line containing information about the asset followed by 4 lines containing information about options. Composite assets are represented in the same way as simple assets except that they also specify a count of children and are followed by a definition for each child.

`<service>`

A *service* asset is represented in the same way as a simple asset, except that it cannot have a replication factor or ordering option.

`<name>`

A name may be used as the name of an asset or the base name for a C source file.

`<min> and <max>`

These are the integers representing minimum and maximum replication factors. If they are both 1, there is no replication. *Min* must be > 0 and *max* must be 0 or >= *min*. An asset will be replicated at least *min* times and at most *max* times. If max is 0, there is no fixed maximum and the asset is replicated as many times as necessary to use all available processors. If *max* = *min*, an asset will be replicated exactly *min* times.

`<order>`

This flag indicates whether a replicated asset's return values are returned in the order that the assets were called (ORDERED) or in the order that they finish (UNORDERED).

`<debug>`

This flag indicates whether an asset should be compiled using debug flags (DEBUG) or not (NDEBUG). It may also be used to turn the debugger on and off for each asset.

`<opt>`

This flag indicates whether an asset should be compiled with optimization off (NOPTIMIZE) or on (OPTIMIZE).

`<simpType>`

The type of a simple asset must be *individual* or *representative*.

`<compType>`

The type of a composite asset must be *line*, *department* or *division*.

`<childcount>`

Each composite asset has a *receptionist* and one or more children. The *receptionist* is not explicitly represented in the graph file. Each child asset is represented in the graph file.

`<options>`

Four lines give options for compiling, linking and executing each asset and all four lines must appear. If an option does not apply to an asset, the rest of the line is left blank. The options are treated as character strings by the interface. That is, they will not be parsed but will be passed to the *Enterprise* executive in the form that they are entered by the user. CFLAGS gives a list of compile flags to use when compiling the asset. They are appended to the compile command by the executive. EXTERNAL gives a list of external modules or libraries to be linked with an asset. They are appended to the link command by the executive. INCLUDE gives a list of machines that can execute an asset. If the list is present, the machines will be used instead of the machines in the *Enterprise* mach_file. EXCLUDE gives a list of machines that are forbidden to execute an asset. These will be excluded from the list in mach_file.

Appendix B: The Enterprise Event File Format

This appendix describes the format of the *Enterprise* event file using extended BNF notation.

Notation

```
<abcd>*  means 0 or more of <abcd>
<abcd>+  means 1 or more of <abcd>
a|b      means a or b
()       is used for grouping
```

Syntax

```
<eventFile>    ::= <event>*

<event>        ::= # (<sentEvent> | <rcvdEvent> | <doneEvent> | <blockEvent>
                       | <dieEvent>) <evTime> <comment>*

<sentEvent>    ::= (sentMsg | sentReply) <assetName> <assetName> <msgTag>

<rcvdEvent>    ::= (rcvdMsg | rcvdReply) <assetName> <assetName> <msgTag>

<doneEvent>    ::= doneMsg <assetName>

<blockEvent>   ::= block <assetName> <msgTag>

<dieEvent>     ::= die <assetName>

<comment>      ::= <oneLineOfFile>

<assetName>    ::= <assetBase> <assetSuffix>+

<msgTag>       ::= <integer>

<evTime>       ::= <integer>

<assetBase>    ::= <string>

<assetSuffix>  ::= . <integer>
```

Semantics

`<eventFile>`

An `<eventFile>` contains all of the events that were captured for one run of the program. It consists of zero or more event records. The file is used to communicate between the run-time executive and the animation system.

`<event>`

An `<event>` represents the occurrence of one run-time event. Because each event may span multiple lines in the file, each must be prefixed with the # character. Events are generated in

response to actions taken by the user's program. Each event record contains the time at which the event occurred. The sequence of times must be non-decreasing.

`<sentEvent>`

A `<sentEvent>` can be either a `<sentMsg>` or a `<sentReply>`. A `<sentMsg>` is generated by an asset that has sent a message to another asset. A `<sentReply>` is generated by an asset that has previously received a message from another asset and has just sent a reply for this message. In both types, the record contains the name of the sending asset, the name of the receiving asset, and the tag for the message. Following this line is an optional comment. The comment will be displayed in the message when it is expanded by the user during an animation. Each line of comment will be displayed on a separate line in the expanded message.

`<rcvdEvent>`

A `<rcvdEvent>` can be either a `<rcvdMsg>` or a `<rcvdReply>`. A `<rcvdMsg>` is generated by an asset that has received a message from a caller and started to work on the task. A `<rcvdReply>` is generated by an asset that has accessed a reply from a previous call to another asset. In both types, the record contains the name of the receiving asset, the name of the sending asset, and the message tag. The tag must match the tag of a message (for `<rcvdMsg>`) or reply (for `<rcvdReply>`) that was previously sent.

`<doneEvent>`

A `<doneEvent>` is generated by an asset that has finished a task and become idle. If a reply was sent, the asset must generate a `<sentReply>` event before the `<doneEvent>`. The event record contains the name of the asset.

`<blockEvent>`

A `<blockEvent>` is generated by an asset when it tries to access the returned value of a previously sent message and the reply is not yet available. The event record contains the name of the blocking asset and the tag of the message that was sent and has not yet returned.

`<dieEvent>`

A `<dieEvent>` is generated by the run-time executive when it detects that an asset is no longer responding to messages. The event record contains the name of the asset that has died.

`<comment>`

A `<comment>` is a string of characters with embedded spaces, ended by an end of line. It will be displayed when its message is expanded by the user during an animation. The animation system will not process the string in any way. The run-time executive is responsible for building the string before writing it to the event file.

<assetName>

An <assetName> is a string that matches the name of one of the assets in the graph, including its number suffix. An asset's <assetName> is unique within a program, even when replicas are considered. The <assetName>is built by appending its number suffix to the base name assigned by the user. Its base name consists of its parents name and its number suffix consists of a '.' and its replica number. The root asset just appends a '.1' to its base name. Assets that are not replicated have replica numbers of 1. Replicas of replicated assets are numbered left to right within their parent, starting at 1. When an asset is replicated, it becomes a manager for its replicas. The manager takes the place of the original non-replicated asset in the graph and is the parent for the replicas. For example, consider a line of two assets, A and B. B is replicated twice. Then A will have assetName A.1, and B will become a manager with assetName B.1 There will be two replicas of B created that will have the manager B.1 as their parent. They will have assetNames B.1.1 and B.1.2

<msgTag>

A <msgTag> is an integer that uniquely identifies a message. Message tags are used to associate message replies and message blocks with message sends.

<evTime>

An <evTime> is an integer time measured from some arbitrary start time.

Notes

1. The sender is not actually required in a <rcvdEvent> since the tag can be used to find the corresponding <sentEvent> that contains the sender. However, it is more convenient to include it.
2. When a manager forwards a message to a replica, it must ensure that the replica replies to the original sender and not to the manager. However, the event record must be a <sentMsg> from the manager to the receiver so that the animation system animates the message from the shared replica queue to the replica. That is, the sequence of events for asset A calling replicated asset B and its reply must be:

```
sentMsg A.1 B.1 tag1       (A.1 sends to manager)
rvcdMsg B.1 tag1           (manager receives msg)
sentMsg B.1 B.1.1 tag2     (manager forwards msg to B.1.1)
rcvdMsg B.1.1 tag2         (B.1.1 receives message)
sentReply B.1.1 A.1 tag3   (B.1.1 replies to A.1 directly)
doneMsg B.1.1              (B.1.1 is finished executing)
rcvdReply A.1 tag3         (A.1 uses the result)
```