University of Alberta

Enterprise User's Manual
Version 2.2

Paul Iglinski
Randal Kornelsen
Chris Morrow
Ian Parsons
Jonathan Schaeffer
Carol Smith
Duane Szafron

Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
{duane, jonathan}@cs.ualberta.ca

Technical Report TR 94-04
March 1994

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Enterprise User's Manual
Version 2.2

Paul Iglinski
Randal Kornelsen
Chris Morrow
Ian Parsons
Jonathan Schaeffer
Carol Smith
Duane Szafron

Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
{duane, jonathan}@cs.ualberta.ca

Abstract

This document is a user's manual for version 2.2 of the Enterprise parallel programming system. *Enterprise* is an interactive graphical programming environment for designing, coding, debugging, testing and executing programs in a distributed hardware environment. *Enterprise* code looks like familiar sequential code because the parallelism is expressed graphically and independently of the code. The system automatically inserts the code necessary to correctly handle communication and synchronization, allowing the rapid construction of distributed programs.

1. Introduction

One of the design goals of the Enterprise project has been to provide an environment where parallel/distributed programming is as close to the sequential model as possible. Enterprise has no extensions to the C programming language, nor does it require the user to insert system or library calls into their code. All information describing the parallelism of a program is contained in a diagram (*asset* diagram). The user's code is standard C, with the system automatically inserting the additional code necessary to implement the parallel semantics based on the information provided in the asset diagram. This programming model requires new parallel semantics for some familiar sequential constructs and, for implementation convenience or performance considerations,

imposes a few (minor) limitations. This document provides an introduction to programming with Enterprise.

Section 2 describes the assumptions about the execution environment in which Enterprise programs run. Section 3 describes the semantics of writing C code in Enterprise (the *programming* model). Section 4 describes the techniques for specifying parallelism in an application using asset diagrams (the *metaprogramming* model). Section 5 describes the user interface by constructing a complete application. Section 6 describes the differences between sequential C and Enterprise C. Some restrictions in the current implementation of the model are outlined in Section 7, and Section 8 outlines some deficiencies in the model. Section 9 gives a number of performance tips for getting the best runtime performance for Enterprise applications.

2. The Execution Environment

Enterprise programs are assumed to run on a network of workstations with distributed memory. They currently cannot take advantage of shared memory. Consequently, each process has its own address space, and does not have access to any other process' data. Thus, when one process calls another, the caller must provide all the information necessary (through parameters) for the callee to execute the assigned task.

A networked file system is also assumed. Any files used by an Enterprise program should be accessible from any machine taking part in the computation.

3. The Programming Model

In a sequential program, components of a program exchange information through procedure/function calls. The calling procedure, *A*, contains a call to a procedure, *B*, that includes a list of arguments. When the call is made, *A* is suspended and *B* is activated. *B* can make use of the information passed as arguments. When *B* has finished execution, it communicates its results back to *A* via side-effects to the arguments and/or by returning a value (if it is a function). In Enterprise, procedure calls can be made between processes. Since processes are called assets (for reasons described in Section 3), procedure calls between processes are called asset calls. The semantics of an Enterprise asset call are almost identical to function calls.

As with sequential procedure and function calls, it is useful to differentiate between Enterprise asset calls that return a result and those that do not. Asset calls that return a result are called *f-calls* (function calls) and asset calls that do not return a result are called *p-calls* (procedure calls). Conceptually, there is no difference between a sequential procedure call and an Enterprise asset call except for the parallelism.

Assume *A* and *B* are Enterprise assets executing on different processors. When *A* calls *B*, *A* is not suspended. Instead, *A* continues to execute. However, if the call to *B* was an f-call, then *A* would suspend itself when it tried to use the function result, if *B* had not yet finished execution. In Enterprise, an f-call is not necessarily blocking. Instead, the caller blocks only if the result is needed and the called asset has not yet returned. Consider the following example:

```
result = B( data );
/* some other code : A & B executing in parallel */
value = result + 1;
```

When this code is executed, the arguments of *B (data)* would be packaged into a message and sent to *B* (wherever *B* happens to be running). *A* would continue executing in parallel with *B*. When *A* tries to access the result of the call to *B* (*value* = *result* + 1), it blocks until *B* has returned its result. If *B* is defined as a function, but used as a procedure (i.e. the return value is not used), the result is thrown away. This deferred synchronization is called a *lazy synchronous call*. Assuming that *B* does not modify anything through side-effects, the execution semantics of *A* calling *B* is identical in the sequential and parallel cases.

The p-call in the statement:

```
B( data );
/* some other code : A & B executing in parallel */
```

is non-blocking, so that *A* continues to execute concurrently with *B*. Of course in this case, *B* does not return a result to *A*. This form of parallelism is called *purely asynchronous*. Again, assuming no side-effects for *B*, sequential and parallel execution of this code has the same semantics.

The previous examples are intended to illustrate the similarity between programming sequentially in C and in parallel with Enterprise. This close relationship makes it easier to transform sequential programs to parallel ones and allows the user to change parallelization techniques using the graphical user interface, often without making any changes to the code.

3.1. Parameter Passing

Enterprise assets can accept a fixed number of parameters of varying types. All calls to an asset must have the same number and type of parameters. Arrays and pointers are valid as parameters but they must be immediately followed by an additional size parameter that specifies the number of elements to be passed (unnecessary in sequential C). Unfortunately, this restriction is needed because it is not always possible to statically determine the size of the array to be passed. This feature allows users to pass dynamic storage as well as parts of arrays. The data being passed cannot itself contain pointers (which would be meaningless because of the distributed memory).

Enterprise defines three macros for the parameter passing of pointers. The *IN_PARAM*() macro specifies that a pointer should have its values sent from the caller *A* to the callee *B*, but not returned. The macro *OUT_PARAM*() specifies a parameter with no initial value, but one that gets set by *B* and returned to *A*. The *INOUT_PARAM*() macro copies the parameter from *A* to *B* on the call, and copies its value back from *B* to *A* on the return. For brevity in the text, the parameter passing mechanisms will be referred to as *IN*, *OUT* and *INOUT*, respectively.

Consider the following code fragment illustrating an *INOUT* parameter:

```
void A()
{
   int data[100], result;
   . . .
   result = B( &data[60], INOUT_PARAM( 10 ) );
   /* some other code : A & B executing in parallel */
   value = result + 1;
}

int B( data, size )
int * data, size;
{
   int i, sum;

   for( sum = i = 0; i < size; i++ )
   {
      data[i] *= data[i];
      sum += data[i];
   }
   return( sum );
}
```

A would send the 10 elements 60..69 of *data* to *B*. When *B* is finished execution, it will copy back to *A* 10 elements, overwriting locations 60..69 of *data*, as well as returning the sum. Enterprise makes copies of the passed data so that from *B*'s point of view, there is no distinction between a sequential procedure call and a parallel Enterprise call.

If *IN*, *OUT* or *INOUT* is not specified, *IN* is assumed. It is important to observe that this does not preserve the correct semantics of C. In sequential C, *INOUT* is the default since changes to locations that a parameter points to will be visible to the calling routine. It would be a simple matter to modify Enterprise to preserve the C semantics. However, this difference is explicitly there to make users more aware of the cost of *INOUT* parameters. *INOUT* requires parameters to be sent to and from the asset; *IN* and *OUT* require only one-way communication. In effect, *INOUT* causes data to be copied twice, increasing the overhead at runtime. Thus *IN* and *OUT* parameters are to be preferred whenever possible.

OUT and *INOUT* data implement parameter side-effects. Thus they can also be considered part of the return value of the function. In the above example, if *A* accessed *data*[65] before it accessed *result*, it would have to block until *B* returned, just as it would for the result of the call.

Consequently, only p-calls without *OUT* and *INOUT* parameters are purely asynchronous; all other p-calls and all f-calls are lazy synchronous. In the rest of the document, the return value of an asset will refer to the function result or any *OUT/INOUT* parameters.

Enterprise assumes a distributed memory environment; thus there is no shared memory. All data needed by an asset must be passed to it via parameters. This is the area where most programs will require some conversion effort.

3.2. Replication

Enterprise allows users, through the asset diagram, to specify that a procedure or function (asset) can be executed in parallel and replicate it as often as needed (make multiple copies). Although discussion of this feature properly belongs in Section 3, it is mentioned here so that the scope of the examples in this section can be expanded.

If an asset is replicated, then Enterprise creates multiple copies of the process, subject to any constraints supplied by the user. Each call to that asset is queued by Enterprise and sent to the first available idle instance of that asset. In the following code, assume *Square* is an asset and it has been replicated 3 times:

```
void SumSquares()
{
   int i, a[10];
   ...
   for( i = 0; i < 10; i++ )
   {
      a[i] = Square( i );
   }
   ...
}
```

Since the calls to *Square* are f-calls, execution continues until a return value of *Square* is accessed. In this case, *SumSquares* will loop 10 times and invoke 10 concurrent calls to *Square*. Since there are only 3 copies of *Square*, the first 3 calls are immediately sent to be processed, while the remaining outstanding calls are automatically queued. As a *Square* asset completes its work, it is immediately assigned new work, if available.

This example illustrates that a poor replication factor can affect parallel efficiency. If we simplistically assume that each call to *Square* takes the same amount of time, then with 3 copies we would expect one to get 4 pieces of work, and the other two to get 3. The loop is complete once the last piece of work is done, meaning we have to wait for 4 pieces of work to be done. This represents a 10/4 = 2.5-fold improvement over the sequential program. However, if the replication factor is 5, then each process gets 2 pieces of work and the parallel improvement is 10/2 = 5. Thus, increasing the number of assets from 3 to 5 doubles the performance.

3.3. Unordered Assets

When an asset is called, the caller blocks when the return value (function result and/or *OUT/INOUT* parameter) is accessed, if the asset has not yet completed. This implies that the program sees the results of asset calls in the order that the program accesses them. This is the default *ordered* semantics and preserves the semantics of C. Enterprise also supports assets whose results are accessed in the order that they are completed (*unordered*). Consider the following example, where *Square* is an asset that simply returns the square of its argument:

```
void SumSquares()
{
   int sum, i, a[5];

   for( i = 0; i < 5; i++ )
   {
      a[i] = Square( i );
   }
   sum = 0;
   for( i = 0; i < 5; i++ )
   {
      sum += a[ i ];
      printf( "a[%d] = %2d: sum is %2d\n", i, a[i], sum );
   }
}
```

With *ordered* semantics, each *printf* will block until that particular *a[i]*'s value has been obtained, with the following output:

```
a[0] =  0: sum is  0
a[1] =  1: sum is  1
a[2] =  4: sum is  5
a[3] =  9: sum is 14
a[4] = 16: sum is 30
```

If the goal of the computation is to compute the sum, then it does not matter in what order the terms are added; the sum will be same since addition is commutative. Assume that *Square* has been replicated and there are five copies of it running. *SumSquares* will make five calls, without blocking, to *Square*, and only block when it accesses the first return result. With *unordered* semantics, the *printf* will block only until the first *a*[*i*] result is available, even if it is not the one specified by the code. In other words, *a*[*0*] will be assigned the value of the *first* call to *Square* that returns. A sample output might be the following:

```
a[0] =  4: sum is  4
a[1] =  0: sum is  4
a[2] = 16: sum is 20
a[3] =  9: sum is 29
a[4] =  1: sum is 30
```

Note that the output of this program will vary from run-to-run, depending on the timing of when results are returned. However, the final answer (*sum* = 30) will always be the same.

Unordered assets result in non-deterministic results, but increased performance since less blocking occurs. The user should carefully consider the tradeoffs of using *ordered* versus *unordered* asset calls. They are dangerous to use if you do not fully understand the possible side-effects. *Ordered* calls are the default.

3.4. Terminating calls

When an Enterprise asset call returns, all outstanding Enterprise calls made by that asset are canceled. For example, consider asset *A* making several calls to asset *B*. Perhaps one of the results returned by *B* means that *A* has now completed its task and it returns. If there are any calls that have been made to *B* that have not yet returned, a message is sent to *B* to stop the work. *A* will ignore the result returned.

Consider the following simplified *AlphaBeta* tree searching program:

```
int AlphaBeta( lowerbound, upperbound )
int lowerbound, upperbound;
{
   int branch, result;
   ...
   for( branch = 0; branch < treewidth; branch++ )
   {
      result = AlphaBeta( lowerbound, upperbound );
      if( result > lowerbound )
         lowerbound = result;
      if( lowerbound >= upperbound )
         break;
   }
   return( lowerbound );
}
```

The branches are searched sequentially, and when one of the searches returns a value as large as the *upperbound*, search at this node is complete (a cutoff has occurred).

Consider searching all the branches in parallel. The code won't have any parallelism, since the return value of *AlphaBeta* (*result*) is immediately accessed after the call. To introduce parallelism, we have to force each asset call to save its result in a different location:

```
int AlphaBeta( lowerbound, upperbound )
int lowerbound, upperbound;
{
   int branch, result[treewidth];
   ...
   for( branch = 0; branch < treewidth; branch++ )
   {
      result[branch] = AlphaBeta( lowerbound, upperbound );
   }
```

```
   for( branch = 0; branch < treewidth; branch++ )
   {
      if( result[branch] > lowerbound )
         lowerbound = result;
      if( lowerbound >= upperbound )
         break;
   }
   return( lowerbound );
}
```

In the first loop, *treewidth* asset calls to *AlphaBeta* are made, and only in the second loop are the results of the asset calls examined, possibly causing the program to block.

When a cutoff occurs in the parallel version, the routine returns. There may be several outstanding calls to *AlphaBeta*. Since their results are now irrelevant (the search has been cutoff), Enterprise informs those assets to discontinue their work. These assets are now ready to be called again with other work.

There are a few items of interest in this example:

(1) The function is computing a maximum within the range of values *lowerbound* to *upperbound*. Since this computation is commutative, *unordered* assets could be used to increase the concurrency. Note that this might cause a different value to be returned by the parallel version than the sequential one. If more than one branch can cause a cutoff, it is non-deterministic which one will cause the routine to exit.

(2) This code benefits from a parallel implementation, but the improvement can be hard to predict. Consider two cases of nodes where a cutoff occurs. In the first, the cutoff occurs with the first branch. The sequential program will only examine this branch, find the cutoff and return. The parallel version will start all branches in parallel and only stop when a cutoff has been found. Most of the parallel work is wasted, since only one branch had to be evaluated. In the second case, assume that the cutoff occurs with the last branch. The sequential version will look at all branches and only find the cutoff at the very end. Since the parallel version considers all branches in parallel, it will find the cutoff when the search of the last branch returns. There is the possibility for *super-linear* speedup here, since the other branches can be immediately terminated and less work will have been done than in the sequential case.

(3) The sequential program calls *AlphaBeta* with the latest version of *lowerbound*. Although it is not shown in the code fragment given, the *lowerbound* can be used to reduce tree size. Since the parallel calls to *AlphaBeta* all start their computation with the initial *lowerbound*, they may end up doing more work than the sequential program.

The parallelism is obvious in this simple example. Predicting the performance of the parallel program is a difficult task.

4. The Metaprogramming Model

Enterprise's analogy between program structure and organizational structure eliminates inconsistent terminology (pipelines, masters, slaves) and introduces a consistent terminology based on assets in an organization. Currently Enterprise supports the following asset types: enterprise, individual, line, department, division, service, receptionist and representative. Receptionists and representatives are parts of assets that are automatically inserted by Enterprise. Figure 4.1 shows the assets and their icons in Enterprise. Asset icons are used to draw diagrams that specify the parallelism in an Enterprise program.

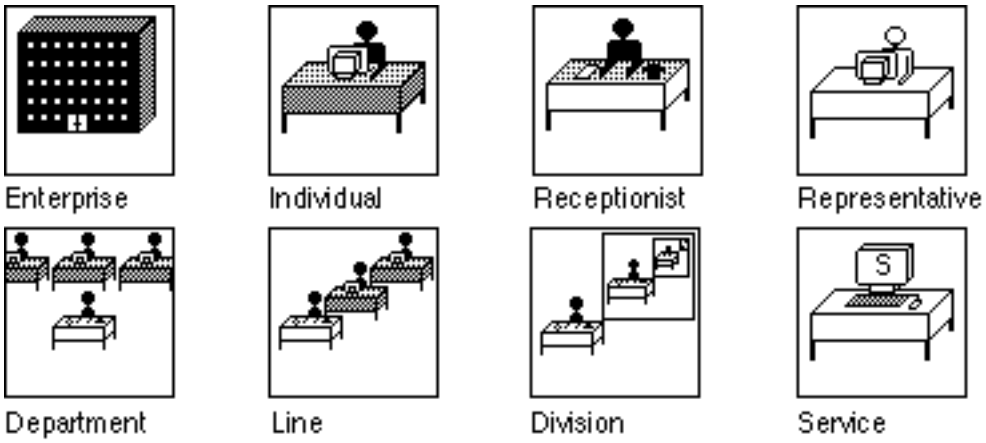


Figure 4.1 Enterprise asset icons.

The user draws an organizational chart that contains assets. The user can expand and collapse composite assets such as lines and departments to show the desired level of abstraction in a hierarchical diagram. A few points about drawing asset diagrams:

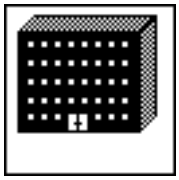
(1) Assets can be combined hierarchically. Wherever it is legal to have an individual, Enterprise allows you to replace it with a composite asset (line, department or division).

(2) Developers can replicate assets so that more than one process can simultaneously execute the code for the same asset. The interface allows the programmer to specify a minimum and maximum (possibly unbounded) replication factor, and Enterprise dynamically creates as many processes as possible, one asset per processor, up to the maximum. Users can explicitly replicate all assets except receptionists and enterprises.

(3) The Enterprise compiler checks to make sure that all assets calls are consistent with the user-specified asset diagram. For example, if *A*, *B* and *C* are in a line, the compiler enforces the rule that *A* must contain a call to *B* (and not *C*) and *B* must call *C* (and not *A*). The only exception to this is a service asset; no asset is required to call a service.

The following sections describe each of the asset types.

4.1. Enterprise



An *enterprise* represents a program and is analogous to an entire organization. Every enterprise asset contains a single component, an individual asset by default, but a developer can transform it into a line, department or division. When Enterprise starts, the interface shows a single enterprise asset which the developer can name and expand to reveal the individual asset inside.

4.2. Individual

An *individual* is analogous to a person in an organization. It does not contain any other assets. In terms of Enterprise's programming component, it represents a procedure that executes sequentially. An individual has source code and a unique name, which it shares with the asset it represents. When an individual is called, it executes its sequential code to completion. Any subsequent call to that individual must wait until the previous call is finished. If a developer entered all the code for a program into a single individual, the program would execute sequentially. Individual assets may call other assets.

4.3. Line

A *line* is analogous to an assembly or processing line (usually called a pipeline in the literature). It contains a fixed number of heterogeneous assets in a specified order. The assets in a line need not be individuals; they can be any legal combination of Enterprise assets. Each asset in the line refines the work of the previous one and contains a call to the next. For example, a line might consist of an individual that takes an order, a department that fills it, and an individual that addresses the package and mails it. The first asset in a line is the *receptionist.* A subsequent call to the line waits only until the receptionist has finished its task for the previous call, not until the entire line is finished.

Consider a graphics program that animates several cartoon characters. For each frame it computes the position of the characters in the animation frame (*Animation*), converts the image to polygons (*Polygon*) and renders it (*Render*), before finally writing it to disk. The code looks similar to the following:

```
void Animation()
{
   int numbcharacters;
   positiontype * characters;

   numbcharacters = InitialPosition( characters );
   for( frame = 1; frame <= maxframes; frame++ )
   {
      Polygon( frame, characters, IN_PARAM( numbcharacters ));
      MovePosition( characters, numbcharacters );
   }
}

void Polygon( frame, characters, numbcharacters )
int frame, numbcharacters;
positiontype * characters;
{
   int numbpolygons;
   polygontype * polygons;

   /* Render each character producing polygons */
   numbpolygons = MakePolygons( characters, numbcharacters,
                        polygons );

   Render( frame,  polygon , IN_PARAM( numbpolygons ));
}

void Render( frame, polygons, numbpolygons )
int frame, numbpolygons;
polygontype * polygons;
{
   /* Render the image and write to disk */
}
```

Animation does not need to wait until *Polygon* has completed its computations on the first frame to start its work on the second. Similarly, *Polygon* does not need to wait for *Render*. The program can be converted to a line with 3 individuals. Figure 4.2 shows the Enterprise asset diagram for this program. In this program, the *Polygon* and *Render* assets are good candidates for replication, to improve performance.

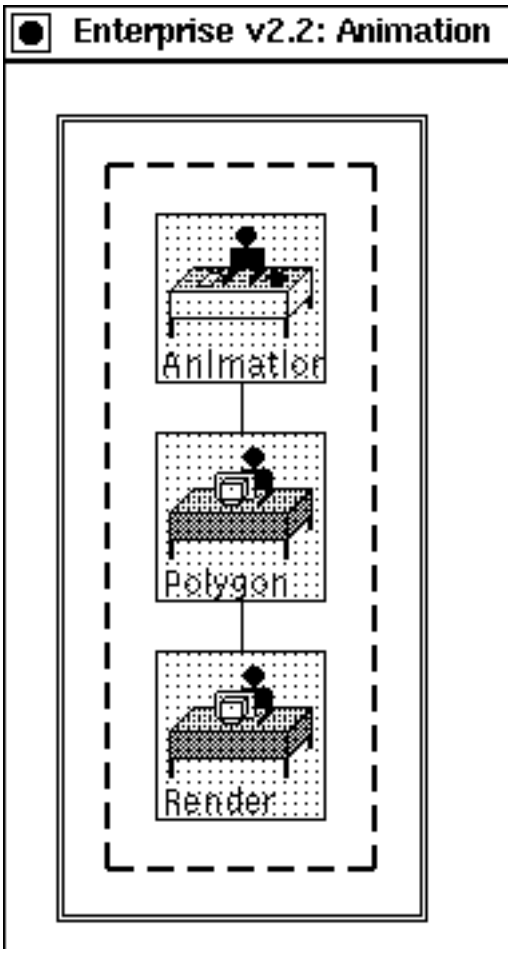
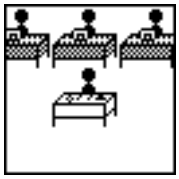


Figure 4.2 *Animation* program as a line.

4.4. Department



A *department* is analogous to a department in an organization. It contains a fixed number of heterogeneous assets and a receptionist that directs each incoming communication to the appropriate asset. Consider the following department example: a customer going to a bank to request a loan. A receptionist listens to the customer's query and, based on the information provided, directs the customer to the appropriate loans officer: mortgages, car loans or business loans. All the bank officials work in parallel. One customer may be served by a mortgage officer while another customer is served by a business loan officer. Of course if a customer wants a mortgage and the mortgage officer is busy, the customer must wait until the mortgage officer is free, unless there are multiple mortgage officers (the mortgage officer is replicated).

As a programming example, consider a program that solves a series of linear equations. There are many different techniques for solving sets of equations, depending on the properties of the matrix. In the following code, several matrices are read in and *MatrixProperty* determines the appropriate routine to use.

```
void LinearEquations()
{
   matrix * m;
   int property, dimension;
   ...
   for( ; ; )
   {
      dimension = GetMatrix( m );
      if( dimension == 0 )
         break;
      property = MatrixProperty( m, dimension );
      switch( property )
      {
          case SPARSE:
            SolveSparse( m, dimension );
            break;
          case TRIDIAGONAL:
            SolveTriDiagonal( m, dimension );
            break;
          default:
            SolveGauss( m, dimension );
            break;
      }
   }
}
```

There are three Enterprise solutions to this problem:

(1) We can enter the code as an individual, *LinearEquations*, with local procedure calls. Enterprise compiles it and runs it sequentially.

(2) *LinearEquations* can be transformed into a department asset, containing the individuals *SolveSparse*, *SolveTriDiagonal* and *SolveGauss* (see Figure 4.3). *LinearEquations* acts as a receptionist, deciding which asset is going to solve each system of equations. When a call is made to one of these individuals, say *SolveSparse*, it will execute in parallel with *LinearEquations*. Since the *Solve* routines do not have return values or *OUT*/*INOUT* parameters, they are p-calls and do not block when there is a subsequent reference to any of their parameters. In the next loop iteration, if *SolveGauss* is called, it executes concurrently with *SolveSparse* and *LinearEquations*. However, if in a subsequent iteration of the loop another call is made to *SolveSparse*, the second call waits until the outstanding call has completed.

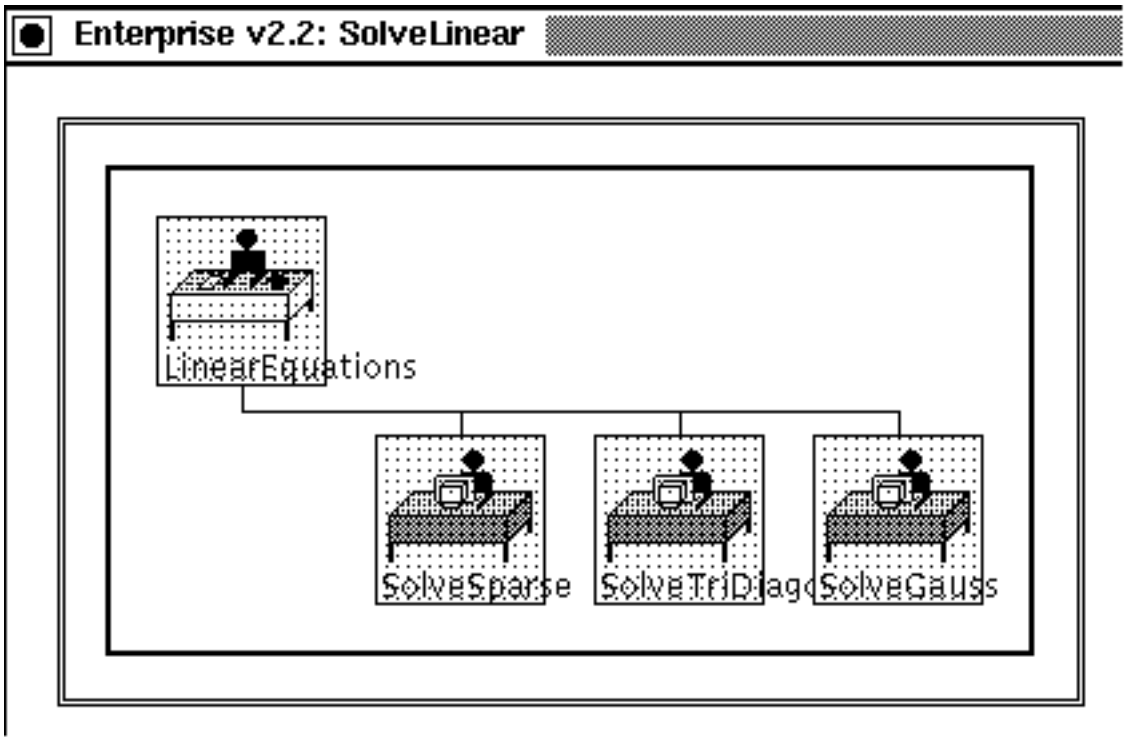


Figure 4.3 *LinearEquations* as a department of individuals.

(3) To reduce the time spent waiting, we can replicate *SolveSparse*, *SolveTriDiagonal* and/or *SolveGauss* to increase the concurrency. If the replication is specified with no maximum replication factor, then Enterprise dynamically assigns all the available processors, as needed, to maximize concurrency.

To have the program run as a department, the user must change the parameters to the *Solve* routines, such as:

```
SolveTriDiagonal( m, IN_PARAM( dimension*dimension), dimension );
```

The matrix *m* is a pointer, and in an asset argument list it must be followed by the number of elements that are to be passed to the asset (*dimension* * *dimension*). The *IN_PARAM* designation is not needed, since this is the default parameter passing mechanism. The *Solve* routines could be modified so that the last parameter is not needed.

It requires only a simple code change to convert the sequential *LinearEquations* individual asset to use departments. No code changes are needed to convert the program from a department of individuals to a department of replicated individuals. Most of the designer's effort is spent deciding on the best asset diagram, and not modifying the source code.

4.5. Division

A *division* contains a hierarchical collection of individual assets among which the work is distributed. Developers can use divisions to parallelize divide-and-conquer computations. When created, a division contains a receptionist and a *representative*, which represents the recursive call that the receptionist makes to the division itself. Divisions are the only recursive asset in Enterprise. Programmers can increase a division's breadth by replicating the representative. The depth of recursion can be increased by replacing the representative with a division. The new division will also contain a receptionist and a representative. The tree's breadth at any level is determined by the replication factor (breadth) of the representative or division it contains. This approach lets developers specify arbitrary fan-out at each division level.

Divisions are a combination of sequential and parallel recursive calls. In the following example, assume *QuickSort* is defined as a division asset.

```
void QuickSort( a, size )
int * a, size;
{
   lower = 0;
   upper = size;
   middle = ( lower+upper ) / 2;
   if( size >= threshold )
   {
      /* Sort each half of the list */
      QuickSort( &a[lower], INOUT_PARAM( middle-lower+1 ) );
      QuickSort( &a[middle+1], INOUT_PARAM( upper-middle ) );
      /* Merge the two sorted lists */
      Merge( a, size);
   }
   else  InsertionSort( a, size );
}
```

For this program, the division should be defined with a breadth of two and we will assume a user-defined depth of two, as shown in Figure 4.4. The first call to QuickSort will divide the list in half. Since each half of *a* is independent of the other half, the recursive calls to *QuickSort* can be done in parallel. These processes associated with the recursive calls have no division children, which means they will be done sequentially. In other words, the calls to *QuickSort* are done either in parallel or sequentially, depending on the resources available at runtime. Since Enterprise inserts code that allows both the parallel and sequential recursive calls to be made, the user can change the breadth and depth of the division without needing to recompile.

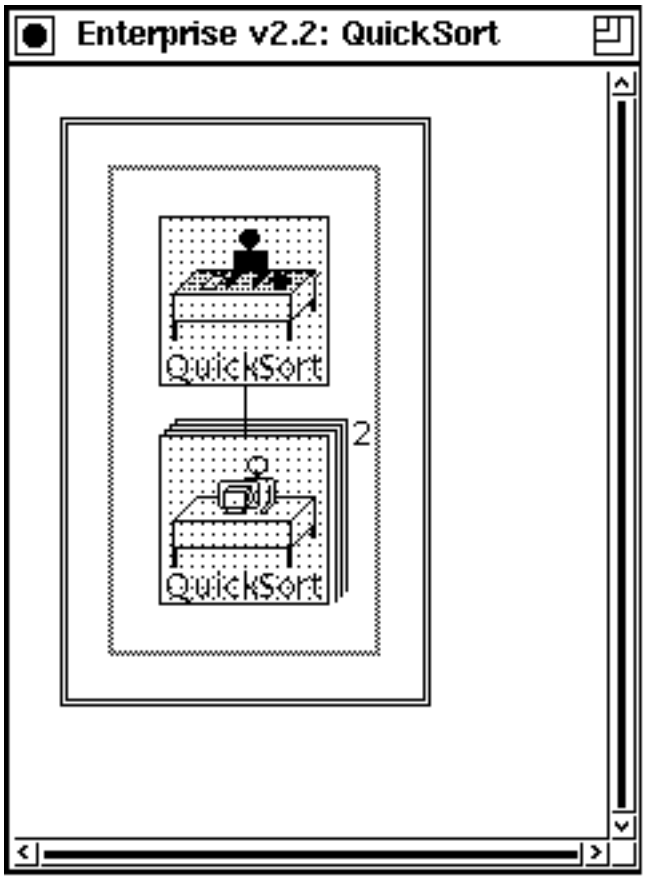


Figure 4.4 *QuickSort* using a division asset.

4.6. Service

A *service* asset is analogous to any asset in an organization that is not consumed by use and whose order of use is not important. It cannot contain any other assets, but any asset can call it. A wall on a clock is an example of a service. Anyone can query it to find the time, and the order of access is not important.

With service assets, it is possible to simulate shared memory. The following code corresponds to part of Figure 4.5. The service asset, *SharedMemory*, accepts two types of calls: one to set the value in shared memory and the other to retrieve the value. Any asset can call *SharedMemory* to set/get the value. Of course, *SharedMemory* could be modified to perform a

more complicated function. For example, a service could be used to queue requests to a shared resource, such as a printer, guaranteeing mutual exclusion for the requests.

```
void AnyAsset()
{
   usertype value;
   ...
   /* Set the value */
   SharedMemory( SET, &value,  IN_PARAM( 1 ) );
   ...
   /* Get the value */
   SharedMemory( GET, &value, OUT_PARAM( 1 ) );
   ...
}

void SharedMemory( access, value, size )
int access, size;
usertype * value;
{
   usertype SharedValue;

   if( access == SET )
      SharedValue = *value;
   else if( access == GET )
      *value = SharedValue;
   else  printf( "SharedMemory: illegal access\n" );
}
```

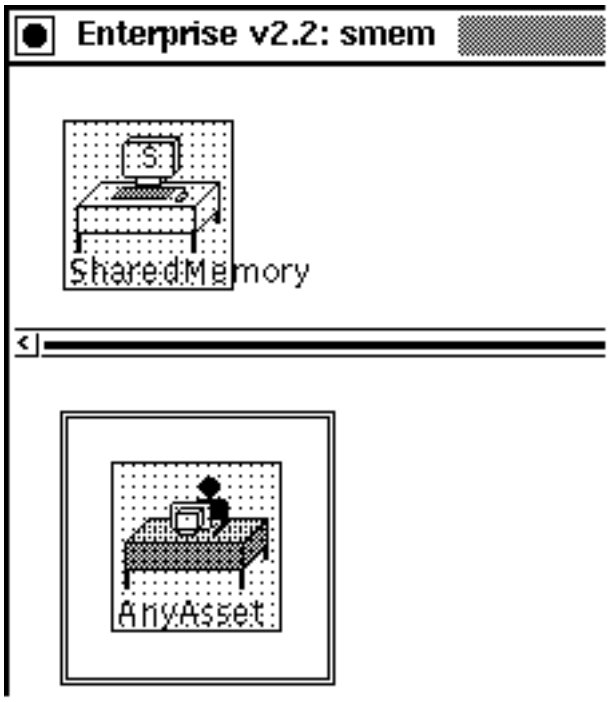


Figure 4.5 Simulating shared memory with a service.

5. The User Interface

This section contains a complete example of building a distributed program using Enterprise. Parallelizing an application using Enterprise involves the following main steps:

(1) Divide the problem into assets and select one of Enterprise's parallelization techniques for each asset.

(2) Build the asset graph.

(3) Enter the source code for the assets.

(4) Compile the assets and fix any syntax errors.

(5) Run and debug the program, fixing any logic errors

(6) Tune the program's performance.

5.1. The Problem

Consider a program called *AlphaBeta* that builds a search tree in a recursive, depth-first manner to a prescribed depth. A user-defined distribution assigns random values to leaf nodes. The parallel version uses the principal variation splitting algorithm, which recursively descends the leftmost branch of the tree searching the siblings in parallel while backing up the result.

The complete source for AlphaBeta can be found in the Enterprise release. A simplified version of the program is presented here. The program has four major procedures: *AlphaBeta*, *Pvs*, *Nsc* and *Draw*. They have the following functionality and pseudo-code:

- The main procedure, *AlphaBeta*, loops calling *Pvs* while we want to do searches.

```
AlphaBeta()
{
   while( do_search )
   {
      Draw( INIT );
      Pvs( 0, -INFINITY, +INFINITY, searchdepth );
   }
}
```

- *Pvs* (Principal Variation Splitting) recurses down the left-most branch of the search tree, doing it sequentially and the remaining branches in parallel.

```
Pvs( branch, alpha, beta, depth )
{
   /* Recurse until it no longer pays to do things in
       parallel. Switch to Nsc to search the tree sequentially */
   if( depth <= granularity )
      return( Nsc( branch, alpha, beta, depth ) );

   /* Move down the tree */
   Descend( branch );
```

```
   Draw( NEW_DEPTH, depth, branch );

   /* Find out the children of this node */
   w = Generate( branches );

   result[1] = -Pvs( branches[1], -beta, -alpha, depth-1 );
   if( result[1] > alpha )
      alpha = result[1];
   if( alpha >= beta )
   {
      Ascend( branch );
      Draw( FINISHED_DEPTH, depth, branch );
      return( alpha );
   }

   /* Search remaining branches in parallel. */
   for( i = 2; i <= w; i++ )
   {
      Draw( PARALLEL_BRANCH, depth, i );
      result[i] = -Nsc( branches[i], -beta, -alpha, depth-1 );
   }

   /* Find the maximum of the values returned */
   for( i = 2; i <= w; i++ )
   {
      value = result[i];
      Draw( PARALLEL_RETURN, depth, i );
      if( value > alpha )
         alpha = value;
   }

   /* Move back up the tree */
   Ascend( branch );
   Draw( FINISHED_DEPTH, depth, branch );
   return( alpha );
}
```

- *Nsc*, sequential alpha-beta search using the negascout enhancement. Search the first move with the full (*alpha*, *beta*) window, and successor moves with a minimal window (*alpha*, *alpha*+1). Since the first branch is best most of the time, this allows us to prove a branch inferior with less cost. In the event that the search returns a score > *alpha*, then this branch is better and we have to research it to get its true value.

```
Nsc( branch, alpha, beta, depth )
{
   /* Calls to Draw are not shown */
   /* Move down the tree */
   Descend( branch );

   if( depth == 0 )
   {
      /* Evaluate this node - get its deterministic random
          number */
      result = Evaluate();
```

```
      Ascend( branch );
      return( result );
   }

   /* Find out the children of this node */
   w = Generate( branches );

   for( i = 1; i <= w; i++ )
   {
      if( i == 1 )
         /* First child is special - search with the full    */
         /* window.*/
         result = -Nsc( branches[1], -beta, -alpha, depth-1 );
      else
         /* Search using a minimal window (difference in
         /* bounds is 1).*/
         result = -Nsc( branches[i], -(alpha+1), -alpha, depth-1 );

      /* Result was outside the window - research to get
          accurate value.*/
      if( result > alpha && result < beta )
         result = -Nsc( branches[i], -beta, -result, depth-1 );

      /* Improved the bound? */
      if( result > alpha )
         alpha = result;

      /* Cutoff? */
      if( alpha >= beta )
         break;
   }

   /* Move back up the tree */
   Ascend( branch );
   return( alpha );
}
```

- *Draw* writes current search activity to a separate program which will graphically display the search activity. A separate program is used to avoid conflicts between X windows event loops and communications kernel message passing.

```
Draw( type, parameters )
{
   if( FIRST_TIME )
   {
      /* create X process */
   }
   /* Send information to display */
}
```

5.2. Selecting a Parallelization Technique

Examining the structure of the *AlphaBeta* program reveals two places where large quantities of work may be generated. The first place is the loop in *Pvs* which calls *Nsc*, the second is the loop in *AlphaBeta*.

Pvs uses *Nsc* to calculate results. Inside of the main loop in *Pvs*, no dependencies exist between calls to *Nsc*. This means that the calls to *Nsc* can be made in parallel. This structure is easily expressed by a department asset. *Pvs* is the receptionist inside of the department, and *Nsc* is the other asset inside of the department. The *Nsc* asset is replicated to allow for parallel calls to the *Nsc* function.

AlphaBeta calls *Pvs*. *Pvs* or *Nsc* will then eventually call *Draw*. Calls of *Pvs* from *AlphaBeta* are required to be sequential to simulate a game playing environment. However, there is no reason why *AlphaBeta* has to wait for *Pvs* to return before starting the next iteration of the loop. A similar relationship exists between the *Pvs*/*Nsc* and *Draw*. *Pvs*/*Nsc* do not have to wait for *Draw* to finish before starting the next iteration of the loop inside of *Pvs*. This kind of relationship is called a pipeline in the literature, and represented by a line asset in Enterprise.

5.3. Building the Asset Graph

Building the asset graph is done from the Enterprise user interface. When Enterprise is installed, it sets up a directory structure in a sub-directory specified by the user. To run Enterprise the user must make that directory the current directory and enter the command `'st80'`, which starts a Smalltalk interpreter. Once Smalltalk starts, the user must evaluate the expression `'Enterprise open'`. This can be done by typing it into a workspace, selecting it, and then selecting `doit` from the middle mouse button menu. The user interface will then appear as shown in Figure 5.1.

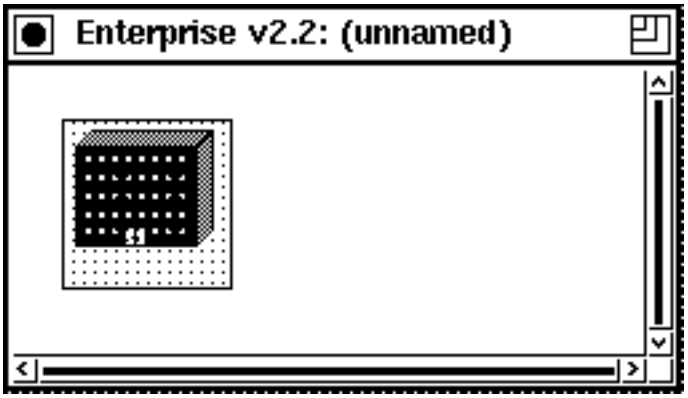


Figure 5.1 A new Enterprise program.

The program canvas initially contains a new enterprise asset. To build the graph we modify this asset. The first step is to name the program using the context-sensitive asset menu. Moving the cursor so it is over the enterprise asset and pressing the middle mouse button will display the menu, which contains the single choice *Name*. Moving the cursor to *Name* and releasing the button will display a dialog box. Typing the name *AlphaBeta* into the dialog box and pressing *ENTER* names the program, with the result shown in Figure 5.2.

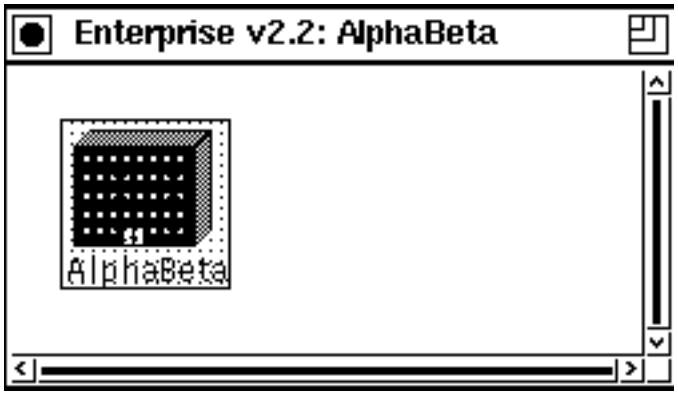


Figure 5.2 A program named AlphaBeta

The asset menu now changes to contain the choices *Name* and *Expand*. Selecting *Expand* from the menu will expand the enterprise asset to reveal the single individual it contains, giving the view shown in Figure 5.3. Note that the individual has been given a default name of *unnamed*.

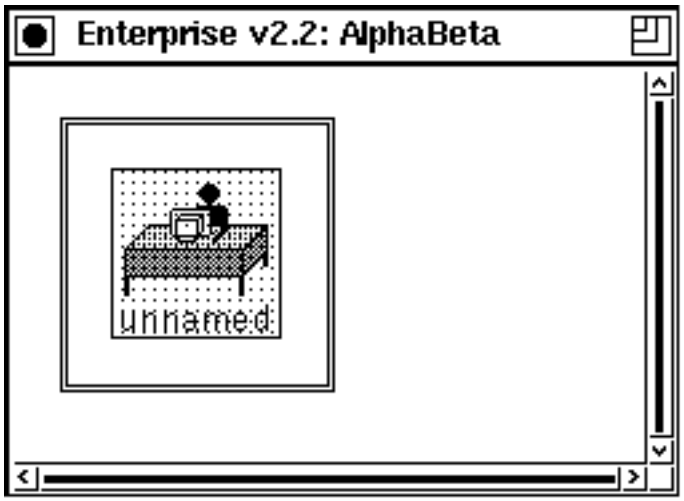


Figure 5.3 An enterprise containing one individual.

Selecting *Line* from the asset menu for the individual will change the individual to a line. Naming the line *AlphaBeta* by selecting *Name* from the line's asset menu gives the view shown in Figure 5.4. The icon now represents a line, and the numeral 2 indicates that the line currently consists of a receptionist and one individual (this is the default).

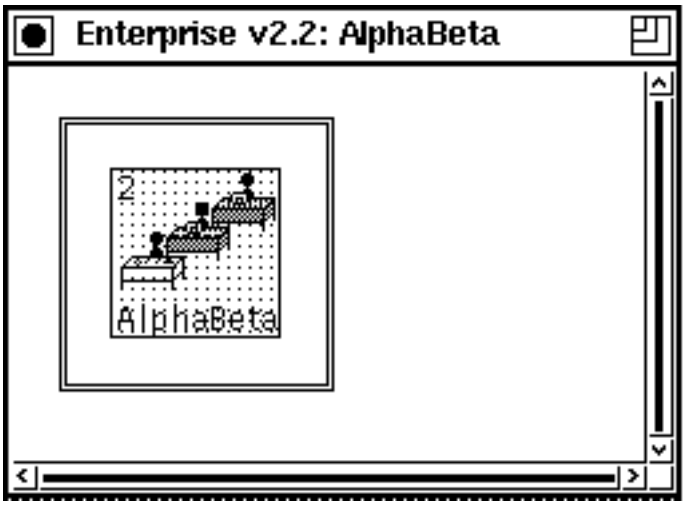


Figure 5.4 A program containing a line.

The line can be expanded by selecting *Expand* from its asset menu. Doing this reveals a receptionist named *AlphaBeta* and an unnamed individual as shown in Figure 5.5.

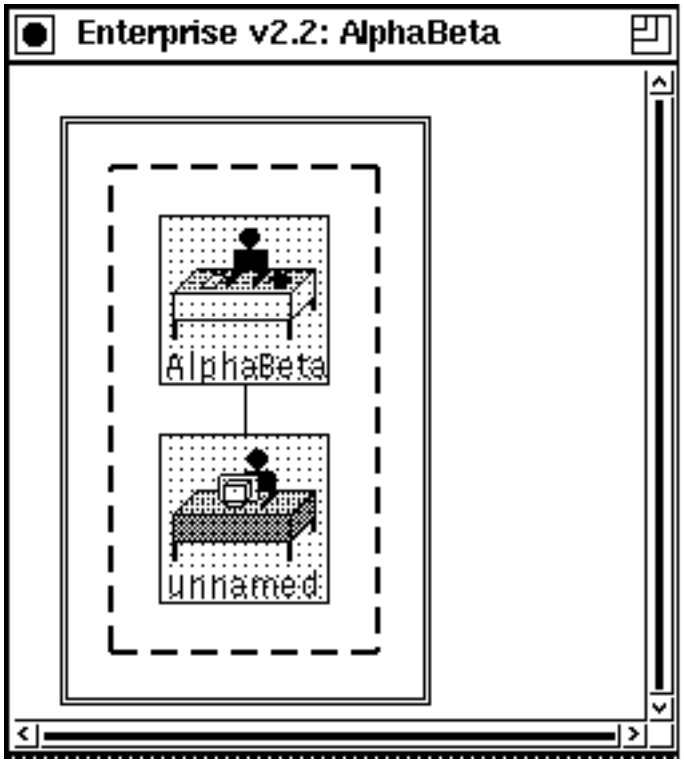


Figure 5.5 An expanded line asset.

The double line represents the enterprise, the dashed line represents the line, and the icons represent the receptionist and individual. Clicking inside the enterprise rectangle but outside the line rectangle will display the asset menu for the enterprise asset. Clicking inside the line rectangle

but outside either of its component's icons will display the asset menu for the line. Clicking on the icons for the receptionist or individual will display their respective asset menus.

The second component of the line can now be named `Pvs` by choosing *Name* from its asset menu. A third component can then be added by selecting *Add After* from *Pvs*'s menu. The new individual can then be named *Draw* using its own asset menu. The resulting graph is shown in Figure 5.6.

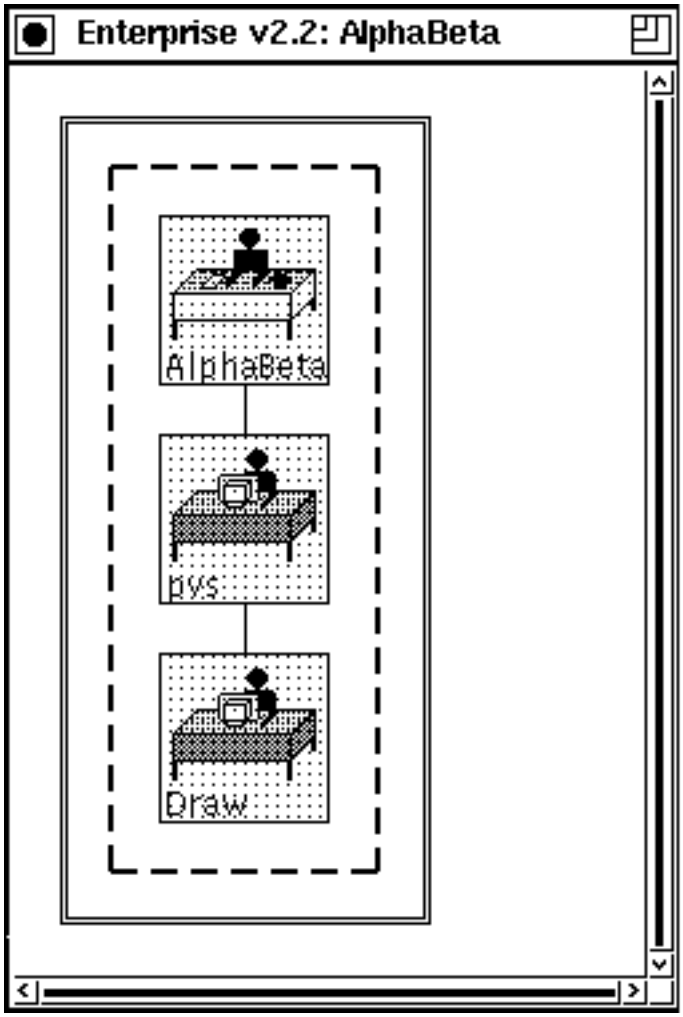


Figure 5.6 A three asset line.

The *Pvs* asset can be changed from an individual asset to a department asset by choosing *department* from its asset menu. The result is shown in Figure 5.7.

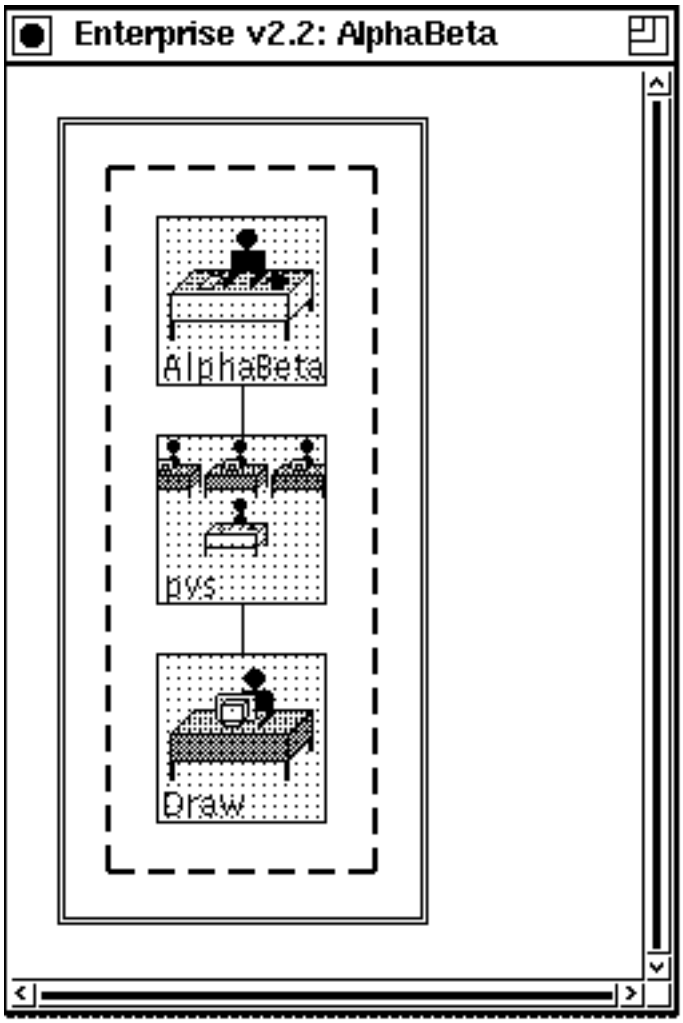


Figure 5.7 A line asset containing a department asset.

The *Pvs* asset can now be expanded by choosing *expand* from its asset menu. This will show a department with a receptionist called *Pvs* and an unnamed individual. The individual can be renamed to *Nsc* in the normal manner using its asset menu. The final asset graph for the program is show in Figure 5.8.

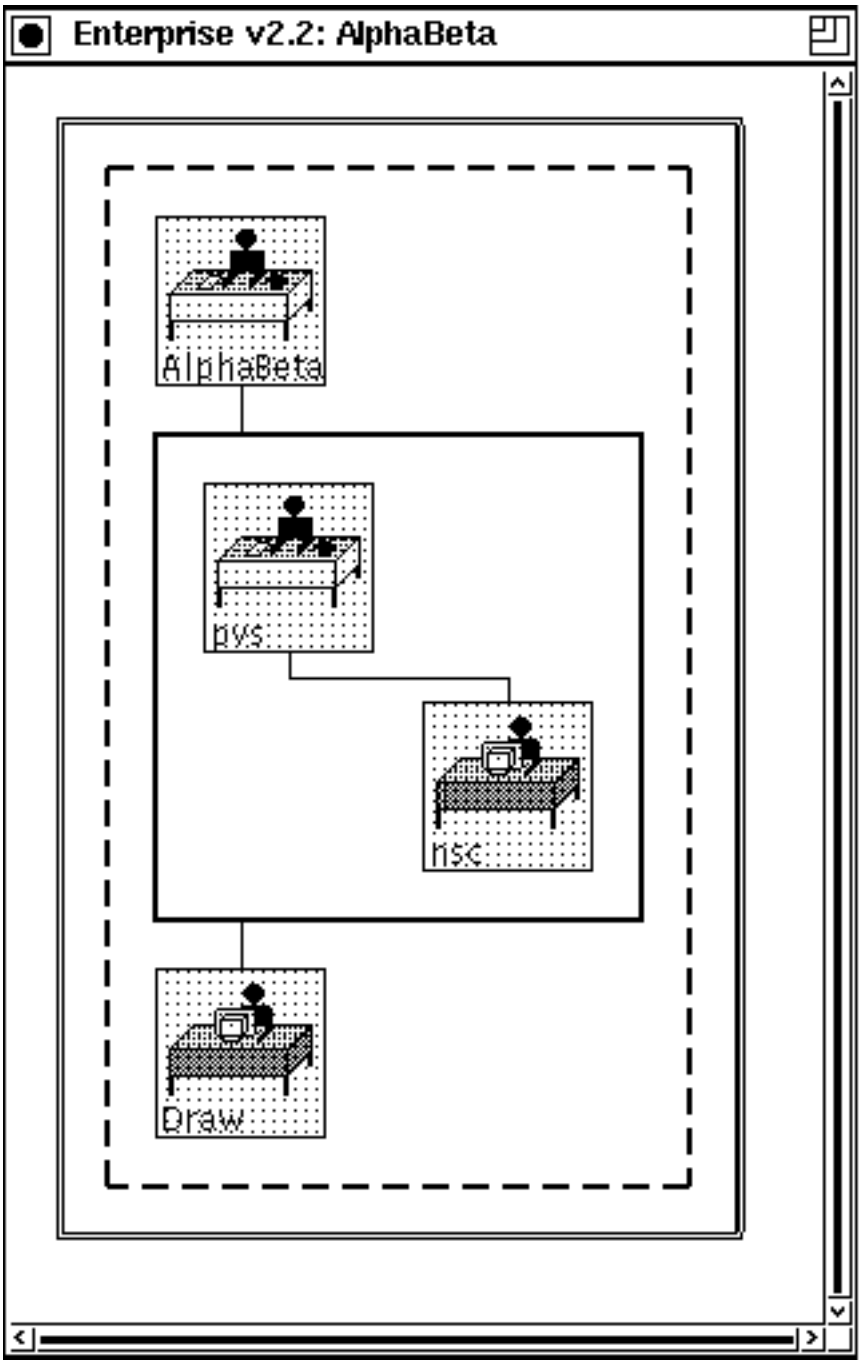


Fig 5.8 Asset graph for AlphaBeta.

5.4. Entering and Editing the Source Code

Enterprise includes an integrated editor for editing asset source code. The editor can be one selected by the user, or it can be the one provided by Enterprise. When the system starts up, the *ENTERPRISE_EDITOR* environment variable is checked. If it is set to *Smalltalk*, then the internal Smalltalk editor is used. If it has some other value, that editor is started in an xterm. If it is not set,

the EDITOR environment variable is checked and used if set. If EDITOR is not set, then the Smalltalk text editor is used.

To invoke the editor, the user selects `Code` from the asset menu for an asset. If source code already exists for the asset, the file is read into the editor, otherwise a new file is created. There can be several editors open at the same time.

The code for an asset consists of an entry function, with the name of the asset, plus any support functions it calls. Procedures that are common to several assets can be placed in a separate file that will be automatically compiled and linked with the asset code. This is done by selecting *Edit User File* from the design view menu and selecting or giving the name of the file to edit. The user can also build a library and have Enterprise link it with the asset code by specifying the library on the *Libraries* line of the compile dialog.

For this example, existing code needs to be organized into files for *AlphaBeta*, *Pvs*, *Nsc* and *Draw*. For each of the assets, we select *Code* from its menu to display an editor. Then we read in the original sequential code and distribute it to the appropriate assets. Note that there is no code associated with a composite asset (line, department or division) so there is no *Code* action in its asset menu.

5.5 Setting Asset Options

A dialog box for setting compilation and runtime options of assets can be opened by selecting *options* from an asset's menu. Figure 5.9 shows an asset options dialog box. The debug/optimize radio buttons and the *CFLAGS* text field specify options for compiling an asset. If output windows or error windows radio buttons are set to *on*, then windows will be opened at runtime to display standard out, and standard error respectively. The *INCLUDE* text field is used to specify a list of machines which this asset may run on. *EXCLUDE* specifies a list of machines which this asset may not run on.

If the asset is a composite asset, then the options specified apply to all assets contained in that asset. If a component asset has options specifically set for it, its options override the options specified in the composite asset that contains it.

In asset options dialog box for *AlphaBeta* shown in Figure 5.9, the *output windows* radio button was set to *on* by clicking with the left mouse button. This will enable all output windows for *AlphaBeta* and all assets that are contained by *AlphaBeta*. When the desired options are selected, press the *OK* button to accept and remember the options.

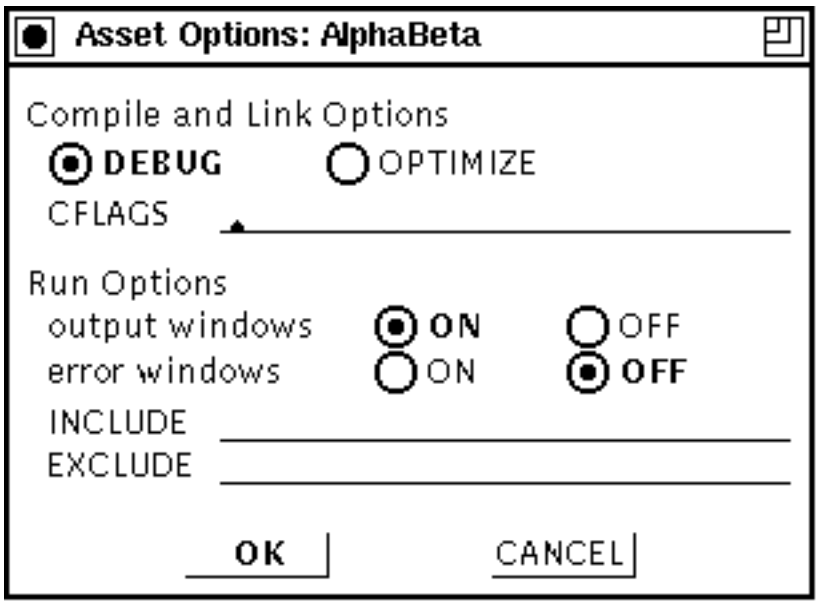


Fig 5.9 Asset options dialog box.

5.6. Compiling Assets and Fixing Syntax Errors

The next step is to compile the code and fix any syntax errors. When the assets are compiled, the Enterprise precompiler inserts the parallelization code, then invokes a standard C compiler and linker to produce the executable program. A compile dialog box can be opened by selecting *compile* from either the design view menu or from an asset menu. If it is started from an asset menu, only the asset is compiled and the linker is not run. If the asset is a composite asset, then all of its components are also compiled. If the compiler is started from the design view menu, the entire program is compiled and linked. The system contains a built-in "make" facility, so only those assets that have changed since the last compilation will actually be compiled. Only one compilation can be active at a time.

The compile dialog box consists of a scrollable text field which will contain output from compilation, a radio switch labeled *verbose* which determines how much compilation output should be displayed, *CFLAGS* and *Libraries* text fields which specify compiler flags, and a number of buttons for starting the compilation. Figure 5.10 shows a compile dialog box.

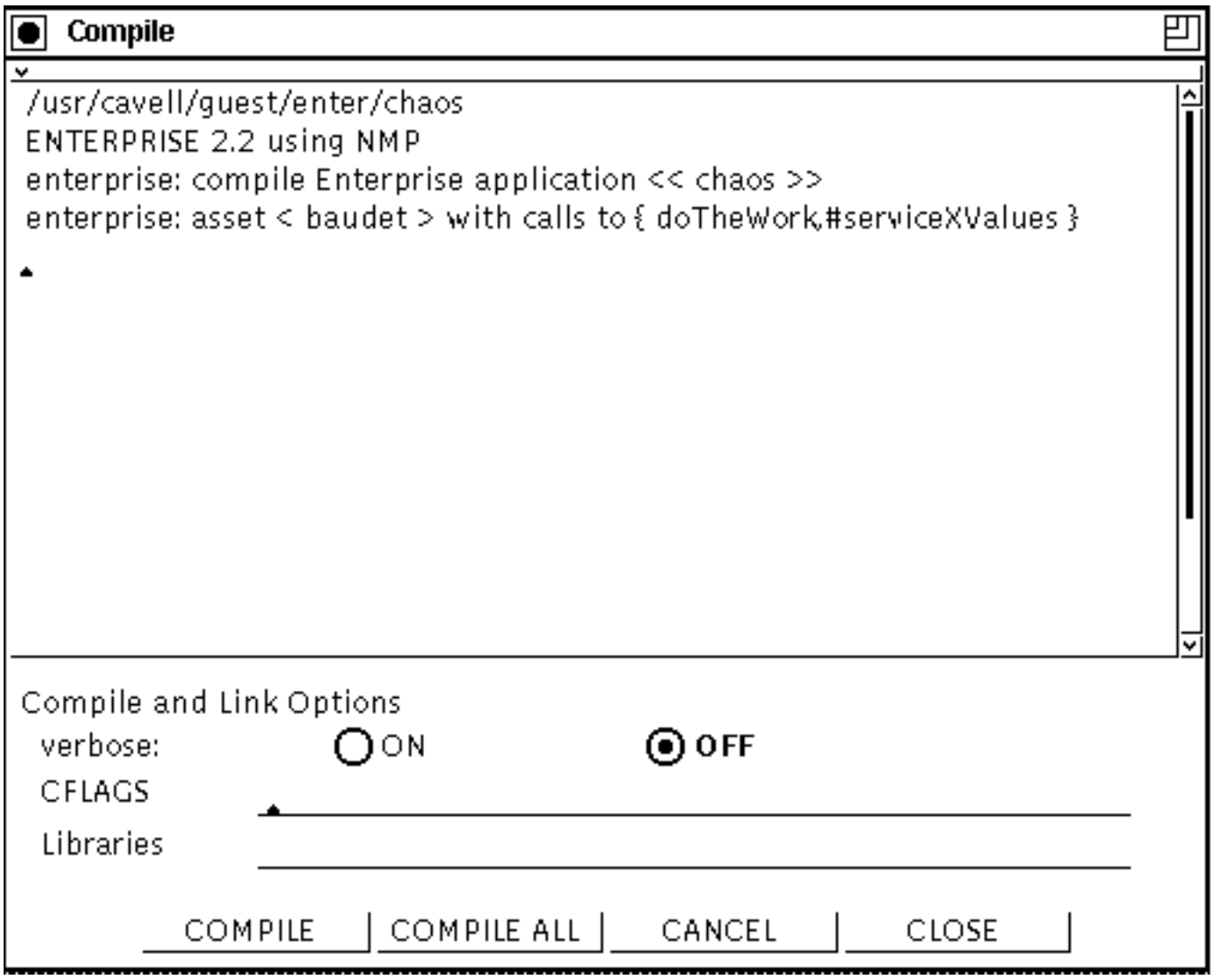


Figure 5.10 A compile dialog box.

If there are errors, the user can leave the compiler window open, invoke editors on the assets, and fix the errors. When the program is being re-compiled, it will use the same compiler window. If there are no errors, the window can be closed to conserve screen space.

5.7. Running the Program

Programs are run using a run dialog box. The dialog can be opened from the design view's menu by selecting *run*. A run dialog box consists of a scrollable text field for displaying runtime output from the enterprise executive, a scrollable list for displaying the list of machines to use while running, a radio switch labeled *event log* for enabling logging of events, text fields labeled *input file* and *output file* for specifying input and output files, a text field labeled *command line*

arguments for specifying runtime command line arguments, a button for starting a run, a button for opening a machine file dialog, a button for canceling a run, and a button for closing the dialog.

In the run dialog shown in Figure 5.11, event logging was enabled by clicking on the *on* event log radio button. Command line arguments were set to *Data/in sunset*, where *Data/in* is the name of an input file, and *sunset* is the name of the machine where we are going to display the graphical output from *AlphaBeta.*

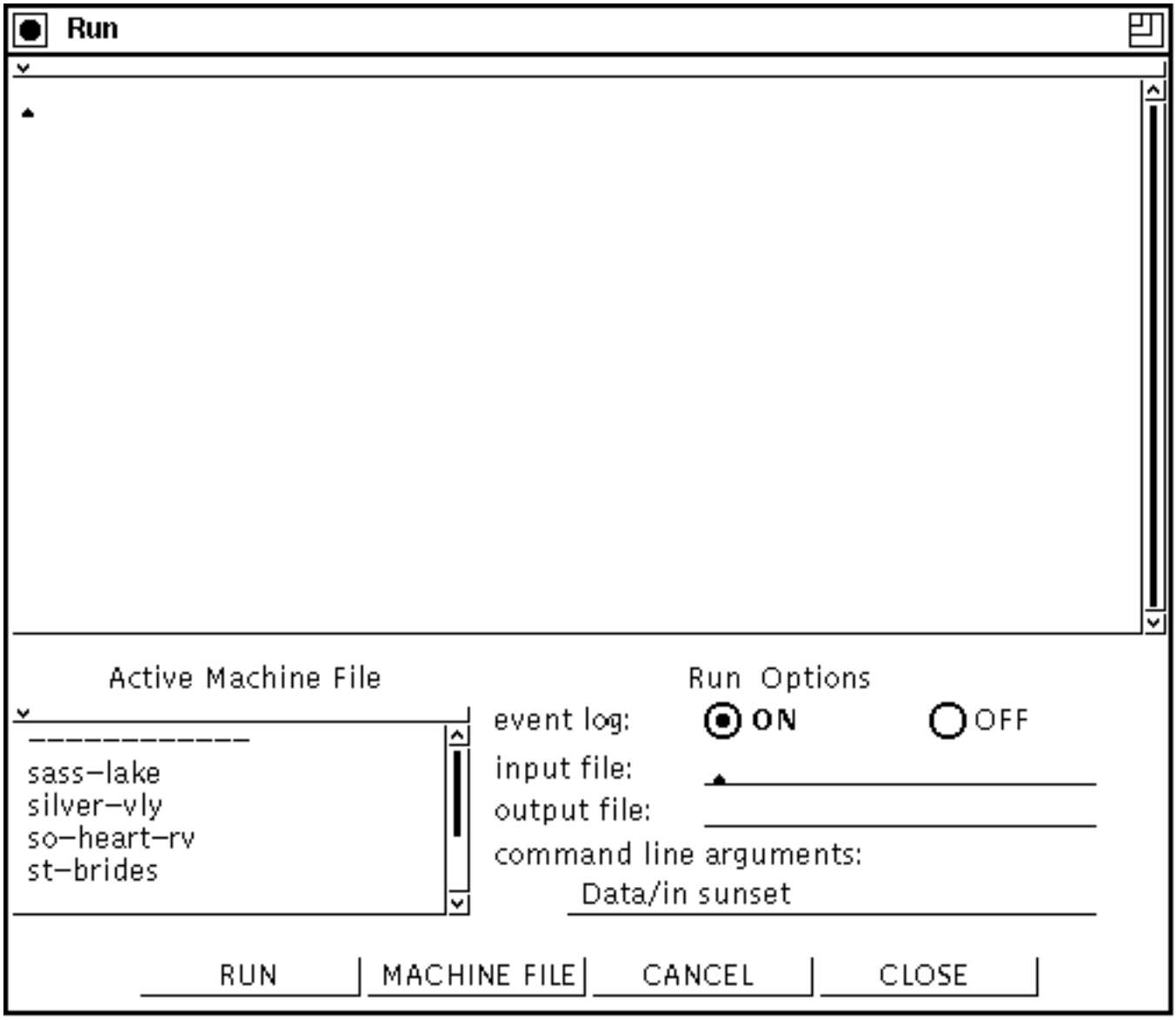


Figure 5.11 A run dialog box.

Before the compiled program can be executed, the system must be told which machines it can use. This is done by opening the machine file dialog using the Machine File button in the run dialog. The machine file editor will then open as shown in Figure 5.12. Here we specify a set of

machines that Enterprise will use to execute assets. The current set of machines are displayed in a list box. Names with #'s in front of them are excluded from use. Typing the name of a machine in the text field below the box, then clicking the *Add* button will add the machine name to the list. Selecting a machine name from the list, then clicking the delete button will delete the machine from the list. Selecting a machine name from the list, then clicking the *include* button will remove the # from the beginning of the machine name. Selecting a machine name from the list, then clicking the *exclude* button will add a # to the beginning of the machine name. Clicking the *OK* button saves the list and exits the editor.

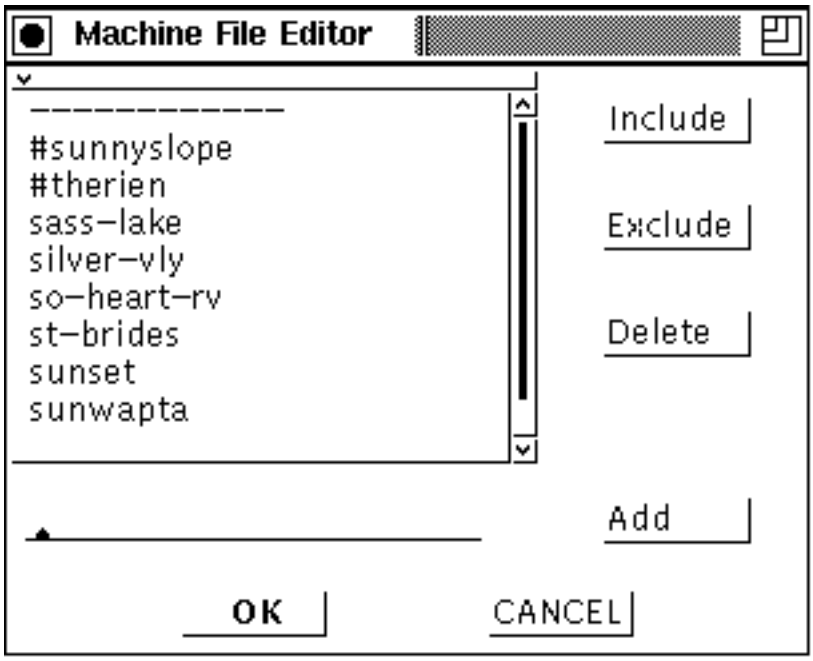


Figure 5.12 A machine file dialog box.

AlphaBeta uses a separate program to display its results. The program must be compiled outside of Enterprise. It resides in the Draw directory under the *AlphaBeta* directory where Enterprise is installed. There is a makefile that will compile the program.

Once the machines have been specified, the program can be executed. This is done by clicking on the *RUN* button in the run dialog. Note that when we set the options in Section 5.5, we elected to have output windows opened for each asset. This means that when the program is run, a window will be created for each asset that will contain any output the asset writes to its standard output file. After the run, the windows are left open until explicitly closed by the user.

The program runs in a sub-directory of the directory where Enterprise resides. Each program has its own directory whose name is based on the name of the program. The program runs from this directory. The sample program reads data from a file. Figure 5.13 shows the resulting screen display.

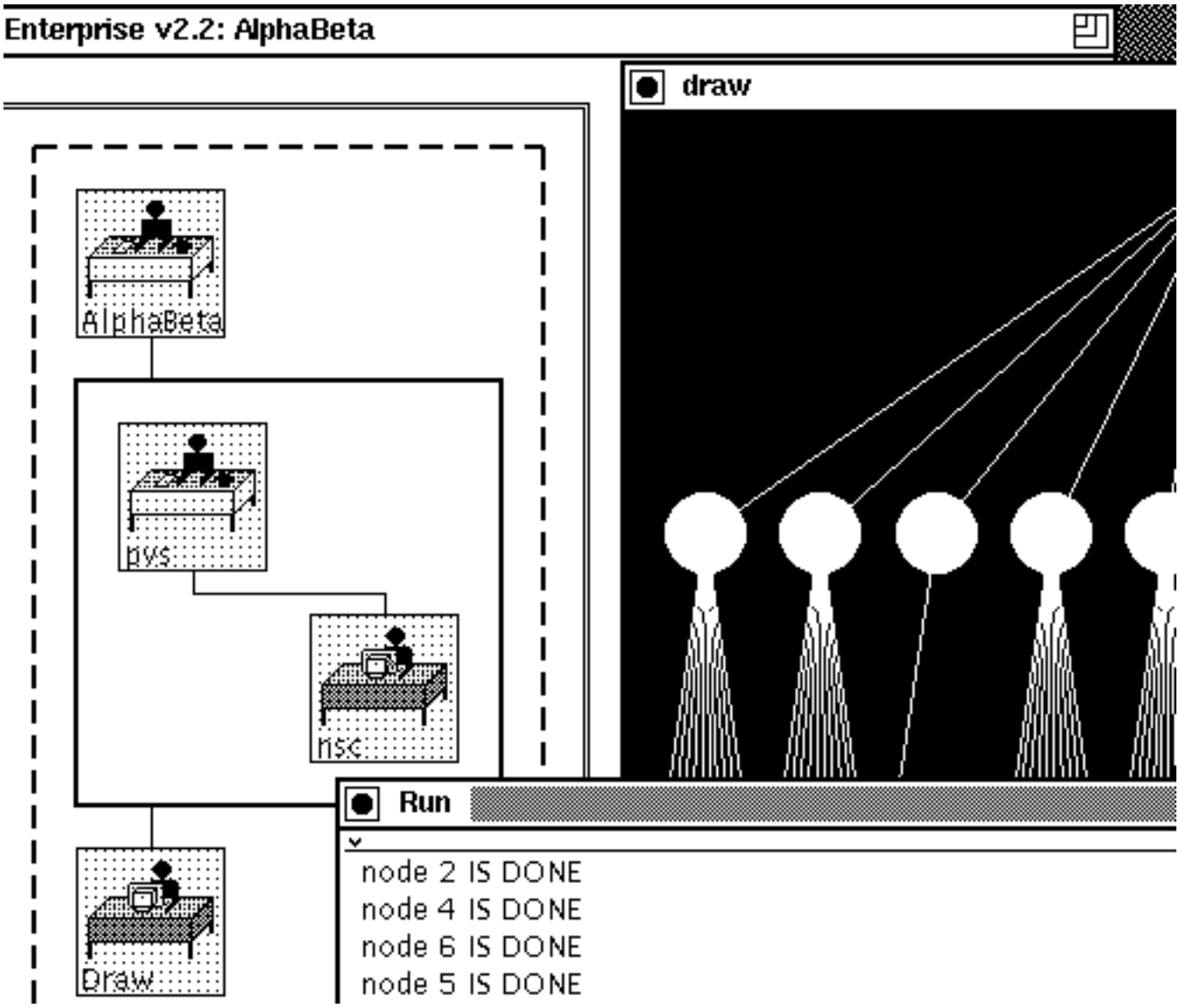


Figure 5.13 Running the AlphaBeta program.

5.8. Performance Tuning Using Animation

When we set the global compile and run options in Section 5.7, we turned on the event logging flag. This caused the program to capture events while it ran and logged them to an event file so the execution can be animated. This allows us to see if there are any performance bottlenecks and to see if the program is making full use of its resources.

Program display is done in the animation view. The view is displayed by selecting *Animate* from the design view menu. After changing views, the program appears as in Figure 5.14. The animation view is similar to the design view in Figure 5.8, but the state of each asset is displayed and there is space to display message queues above the assets and reply queues to their right.

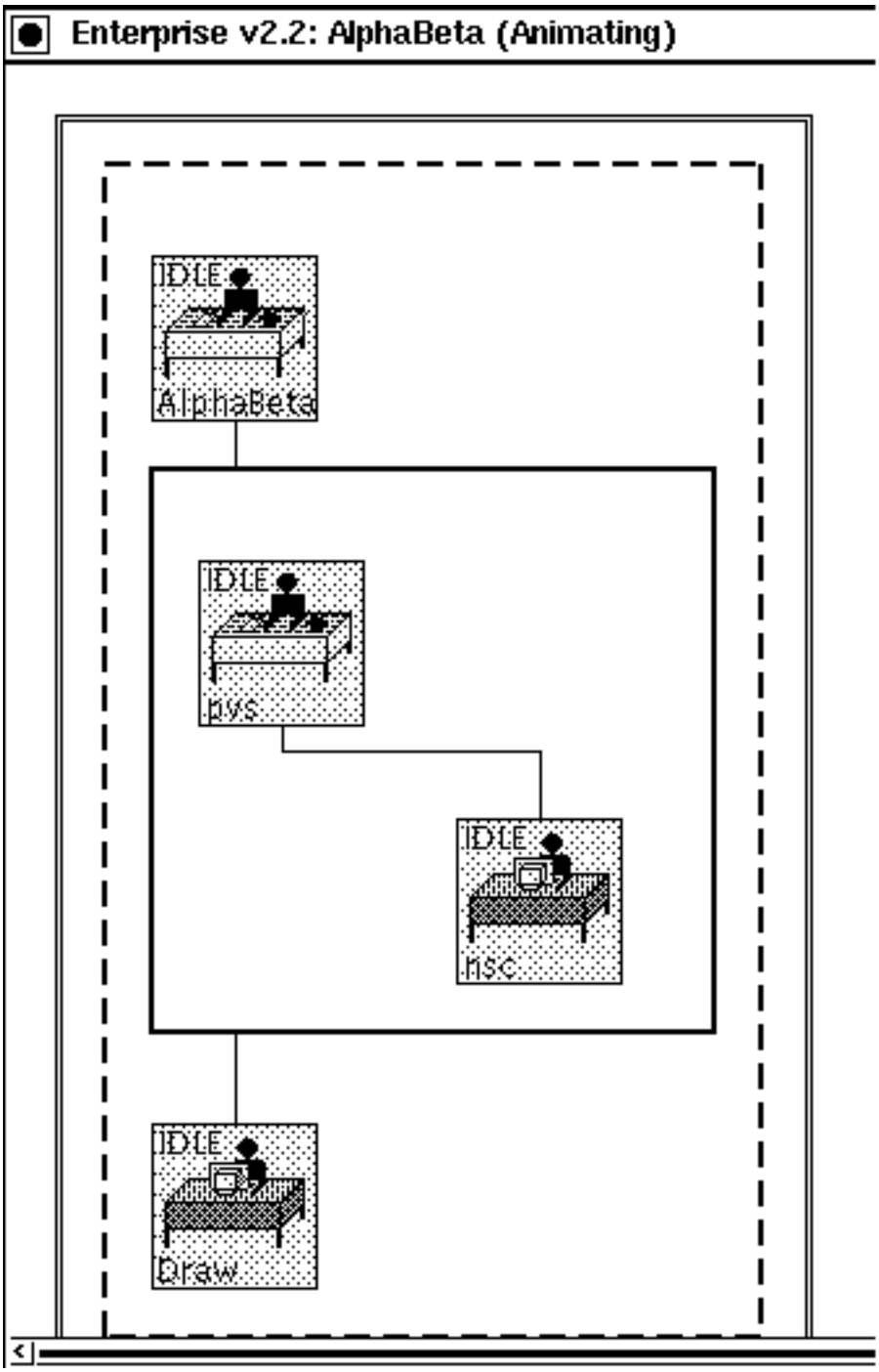


Figure 5.14 The animation view of the sample program.

The menus that appear are different as well. Clicking in the animation view, outside of the assets displays the *animation menu*. If we select *Start* from the menu, the program execution will be displayed, showing messages and replies moving between assets and updating asset states. Eventually, the messages build up in *Nsc*'s input queue as shown in Figure 5.15.

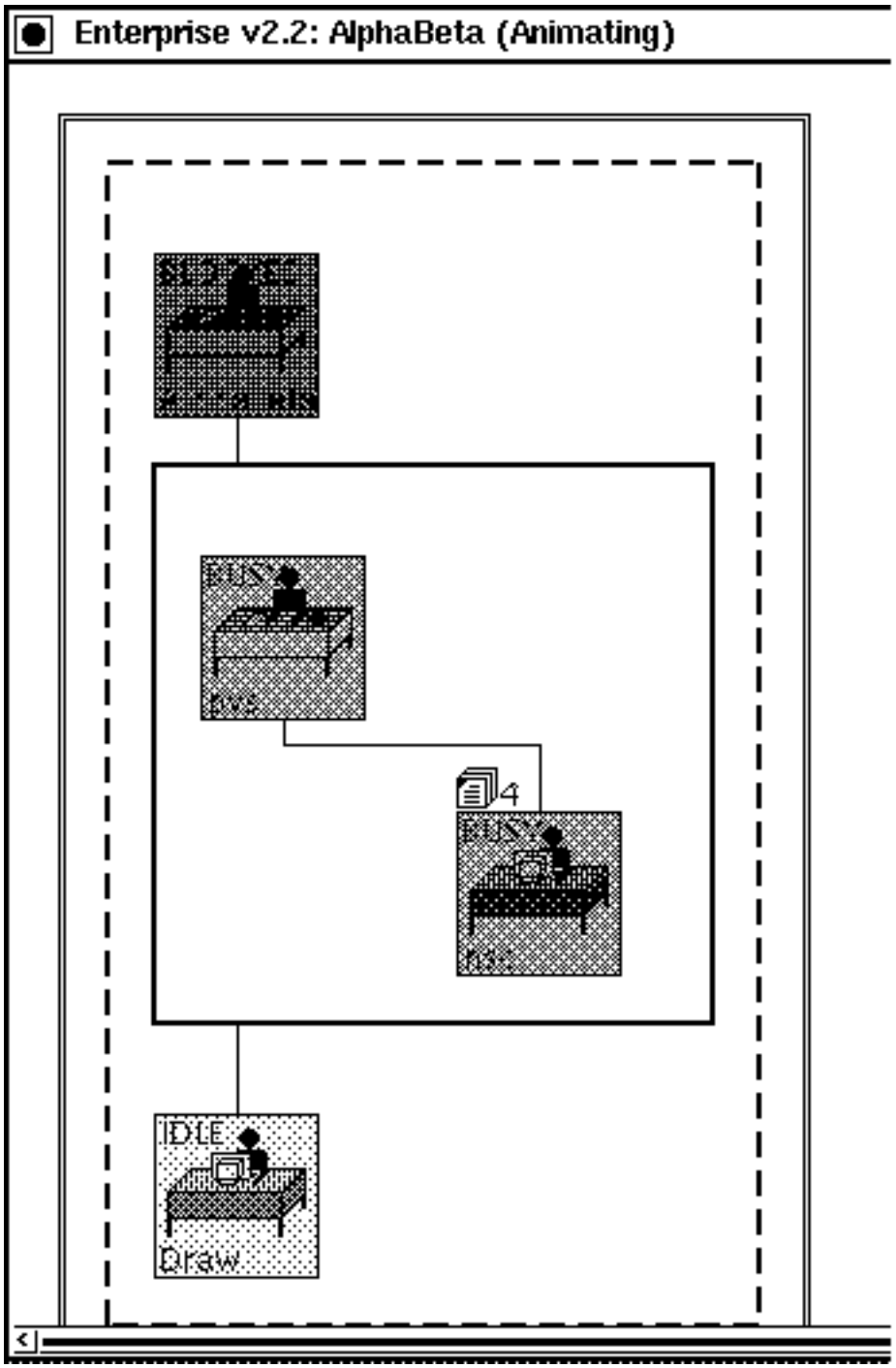


Figure 5.15 A point in the animation.

If the middle mouse button is pressed while the cursor is over a message queue, the context-sensitive menu will display the single entry, *List messages*. Selecting this entry opens an inspector window on the message queue as shown in figure 5.16. Selecting one of the messages in the queue displays the values of the parameters in the C procedure call to *Nsc*.

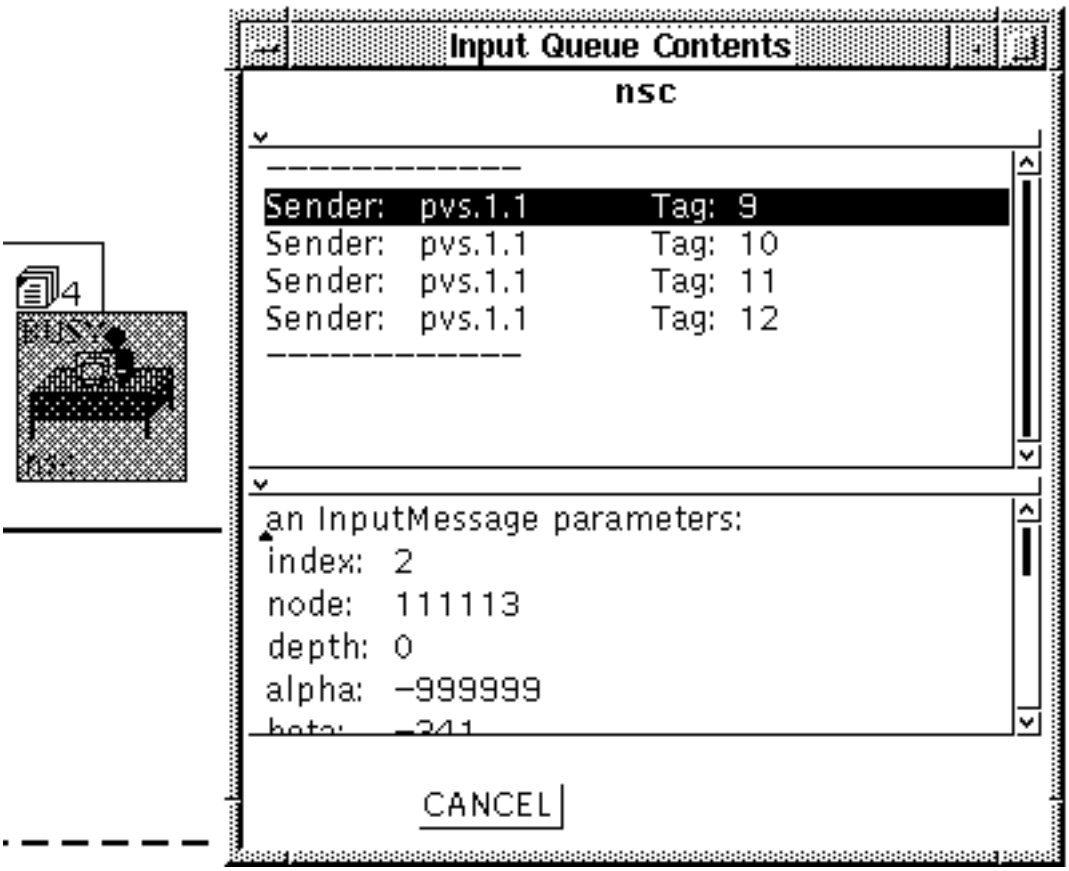


Figure 5.16 A message inspector

Nsc can't keep up with the amount of work being sent to it by *Pvs*. We can try to improve the performance by replicating *Nsc*. To do this, we end the animation by selecting *Stop* from the animation view menu, then switch back to the design view by selecting *Design view*. Once we are in the design view, we select *Replicate* from *Nsc*'s asset menu, causing the replication dialog box to appear as shown in Figure 5.17.

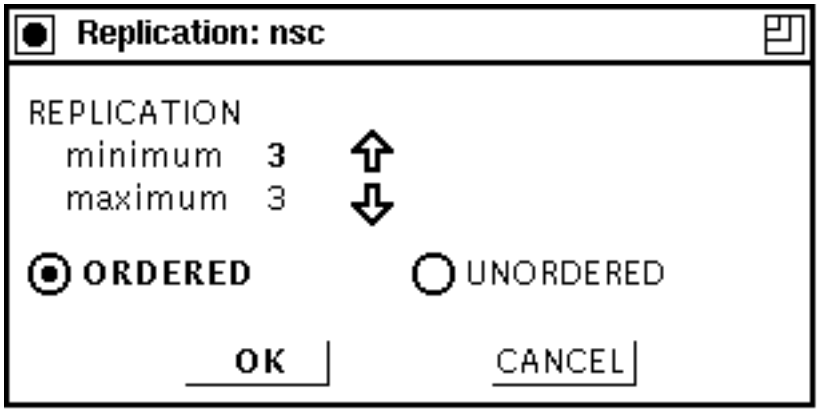


Figure 5.17 The replication dialog box.

We replicate *Nsc* three times by clicking on the up arrow until the maximum replication factor becomes 3. Now we can close the box, and re-execute the program. When we switch to the animation view after re-executing the program, we can see the replicas of *Nsc*. This time when we display the run, we see that all assets keep busy. The resulting animation is shown in Figure 5.18.

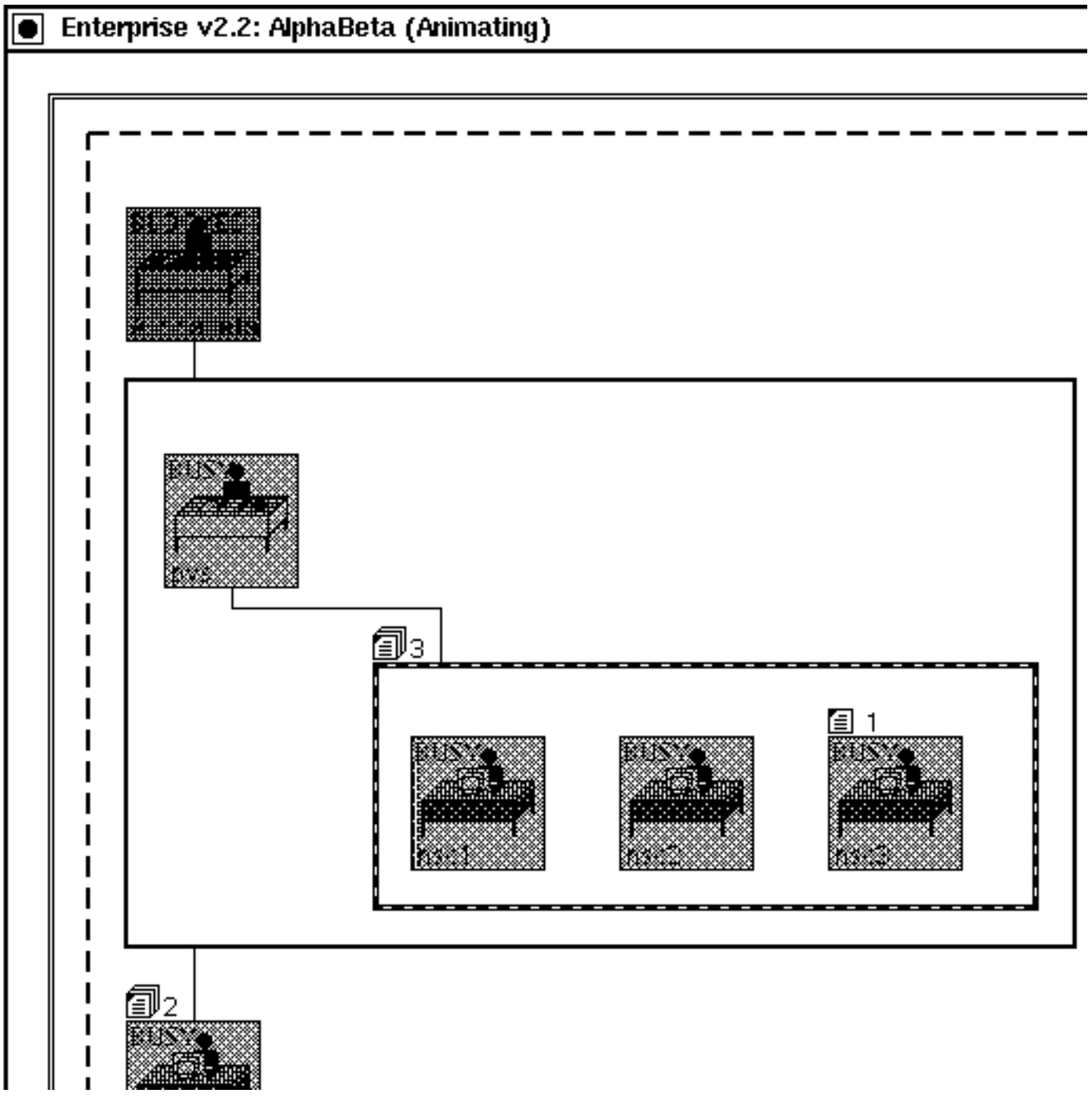


Figure 5.18 Animation view after replicating Nsc.

6. Programming Notes

Programming in C with Enterprise is almost the same as sequential C programming. This section details the differences. Some of these are implementation restrictions that could be eliminated at a future date. In the following, assume *Compute* is an Enterprise asset call.

(1) Pointer parameters in asset calls must be followed by an additional size field indicating the number of elements to pass (not the number of bytes). The size field can be enclosed in one of the macros *IN_PARAM*, *OUT_PARAM* or *INOUT_PARAM*, with *IN_PARAM* being the default. It is the user's responsibility to ensure that these sizes are correct. For example, the following calls are legal:

```
int a[10], * p;
struct example s;
. . .
p = (int *) malloc( 100 * sizeof( int ) );
. . .
result = Compute( a, 10, p, 25, &s, 1 );
result = Compute( a, 10, p, INOUT_PARAM( 100), &s, 1 );
```

All of array *a* and structure *s* are passed to *Compute* as *IN* parameters. The first call to *Compute* passes 25 elements of *p* as an *IN* parameter, while the second call passes all 100 elements as an *INOUT* parameter. It is legal to vary the number of elements passed and the type of the parameter passing mechanism.

(2) Asset calls cannot have a variable number of parameters. The number and type of each parameter is fixed at compile time.

(3) The register designation for variables and parameters is ignored (most compilers ignore it already).

(4) Asset calls can only appear as procedure calls or simple assignment statement function calls. They cannot be part of an expression or a parameter list. For example, the following code two statements are not allowed:

```
sum = Compute( x ) + 3;
printf( "answer is %d\n", Compute( x ) );
```

In both cases no parallelism is achieved, since the return value of *Compute* must be immediately used. The Enterprise compiler will not allow these constructs as a warning to the user that they are probably using the model incorrectly.

(5) Enterprise does not recognize aliasing. Consider the following code fragment:

```
int data[100],
result, *r, *s, b, c;
. . .
r = &result;
s = &data[55];
. . .
result = Compute( &data[50], OUT_PARAM( 10 ) );
. . .
b = *r;
c = *s;
```

In this example, **r* is an alias for *result* and **s* is an alias for *data[55]*. When **r* or **s* is referenced, the program should stop and wait for *Compute* to return. To do this properly would require checking all pointer references to see if they are the object of an Enterprise call or an *OUT*/*INOUT* parameter. The compiler could be modified to do this, but the cost of correctly handling this is too high, both at the implementation level and, more importantly, may have a significant impact at runtime.

(6) A valid Enterprise program cannot have an asset named *main*.

(7) The first asset in a program must have argc/argv declarations. The asset looks as follows:

```
FirstAsset( argc, argv )
int argc;
char ** argv;
{
   . . .
}
```

Currently this is not enforced, but you will discover it (painfully) at runtime.

(8) The "`" character is not allowed in strings. Enterprise uses this character to escape some items. The compiler does not check for this.

(9) References to results from asset calls are not allowed in *do*, *while* or *for* loop statements. This restriction is not necessary, but is imposed so that users do not accidentally create an inefficient construct. Consider the following loop:

```
int b[100];
...
for( b[i] = 0; b[i] < 100; b[i]++ )
{
   ...
   a = Compute( b, INOUT_PARAM( 100 ) );
   ...
}
```

There is no way for the Enterprise compiler to determine whether *b[i]* in the *for* loop expression is influenced by the asset call to *Compute*. Each time through the loop, all references to *b[i]* will have

to be checked to see if a result from an asset call is pending that will modify it. Since this can add an unacceptable overhead to the cost of the loop, it is not allowed.

7. Version 2.2 Restrictions

This section details a number of items that either are not working or have been disabled in this version:

(1) Divisions are not enabled.

(2) Replicated assets are not dynamic. Currently only the specified maximum replication factor is used.

(3) When changing the replication of an asset from one to many, or from many to one, the asset must be recompiled.

(4) Currently, a homogeneous network of Sun 4 workstations must be used.

(5) When a function returns, all asset calls that are outstanding are ignored. They should be canceled.

8. Model Deficiencies

A number of deficiencies exist in the current Enterprise model:

(1) There is no way to test if an asset has returned. When you access its return value, you either continue (because the value is available) or block (because it has not yet returned); there is no intermediate state.

(2) Several new asset types would be nice, such as one to support mesh communication, or one to support peer-to-peer communication.

(3) There is no virtual shared memory. All data must be passed through asset calls. The user can simulate shared memory using service assets.

(4) One aliasing problem must be fixed:

```
int s[10];
...
Compute( &s[i], INOUT_PARAM( 1 ) );
NA( s );
```

Compute is an asset and *s[i]* is its return value. The call to a non-asset, *NA*, is really an alias for the entire *s* array. Enterprise currently does not detect this - we will not block at *NA* for *Compute* to return.

9. Performance Tips

This section provides a number of programming suggestions for improving the performance of an Enterprise application. In the examples given, assume *Compute* is an asset.

(1) Carefully consider the method used to pass all pointer parameters. At runtime, *IN* is the least expensive parameter passing mechanism, while *INOUT* is the most expensive. Always try to use the least expensive method possible.

(2) Ensure that asset calls do enough work to offset the overhead of the parallelism (have enough *granularity*). A good rule of thumb is that assets (other than services) should execute for at least one second per call.

(3) Reduce the number of references to asset return values in your program. If a variable is part of an asset's return value, restrict its usage. All references to asset return values must be checked to see if the asset call has completed, even in obvious cases where no check is needed. The Enterprise compiler does not do flow control analysis to eliminate unnecessary checks. Consider the following code:

```
int i, array[10];
...
for( i = 0; i < 10; i++ )
{
   array[i] = 0;
}
i = Compute( array, INOUT_PARAM( 10 ) );
```

In this example, *array* is part of the return value of *Compute*. The compiler generates code so that all occurrences of *array* are checked to see if the asset call has returned. It is obvious in this example that the usage of *array* in the *for* loop has no relation to the usage of *array* in the call to *Compute*, however the compiler does not detect this. The solution is simple - use *array* only for the purposes of the call to *Compute* and references to the return value, and use another name for *array* in the *for* loop. For example, the following would work:

```
int i, * a, array[10];
...
a = array;
for( i = 0; i < 10; i++ )
{
   a[i] = 0;
}
i = Compute( array, INOUT_PARAM( 10 ) );
```

(4) The previous point implies that if a member of an array is returned by a function, or is an OUT/INOUT parameter, Enterprise must generate code for all references to that array to see if they are accessing a returned result. Consider the following example:

```
int data[100], result[10];
. . .
for( i = 0; i < 10; i += 2 )
{
   result[i] = Compute( &data[i*10], OUT_PARAM( 10 ) );
}
. . .
a = result[j];
```

The reference to *result[j]* might refer to the result of any one of the calls to *Compute*, depending on the runtime value of *j*. Enterprise must keep track of all addresses that a call to an asset can modify. Again, minimize references to return values.

(5) Always call assets with the minimum size of data needed. For example, when passing an array to an asset, rather than pass all the elements of the data structure, only pass the range of values that are actually needed.

(6) To maximize the benefit of having an asset run in parallel, the user should ensure that sufficient computing is performed between the asset call and the time the return value from the asset is accessed. The following example illustrates what not to do:

```
int a;
...
a = Compute();
printf("Answer is %d\n", a );
```

In this case, the user might as well have not used any parallelism, since the calling asset must immediately block and await the return from *Compute*.

(7) Two lazy synchronous calls that modify the same memory locations cannot be active at the same time. For example, the following code is legal but would be executed sequentially:

```
int i, data[100], result[10];
. . .
for( i = 0; i < 5; i++ )
{
   result[i] = Compute( &data[i*10], INOUT_PARAM( 20 ) );
}
```

Each iteration through the loop has the side-effect of modifying overlapping regions of data. Since the sequential semantics of this loop impose the ordering constraints that the second call to *Compute* would use the copy of data returned from the first call, Enterprise cannot execute the calls in parallel.

(8) Ensure that the parallelism that you expect in your program is the parallelism that is actually occurring. Use the animation tool to view the parallelism and compare it with your expectations.

10. Feedback

We would appreciate any feedback that you have on the Enterprise environment. We can be reached electronically at: enter@cs.ualberta.ca. However, if you would like to report a bug, you can do this automatically by selecting *Report Bug* from the design view menu. This feature will send us your program electronically so that we can try to reproduce the bug and fix it.