University of Alberta

Enterprise User's Manual
Version 2.4

Paul Iglinski
Steve MacDonald
Chris Morrow
Diego Novillo
Ian Parsons
Jonathan Schaeffer
Duane Szafron
David Woloschuk

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

Technical Report TR 95-02
January 1995

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

Enterprise User's Manual
Version 2.4

Paul Iglinski
Steve MacDonald
Chris Morrow
Diego Novillo
Ian Parsons
Jonathan Schaeffer
Duane Szafron
David Woloschuk

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 2.4 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 is independent of the code. The system automatically inserts the code necessary to correctly handle communication and synchronization, allowing for 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 debugging in replay view. Section 7 describes the differences between sequential C and Enterprise C. Some restrictions in the current implementation of the model are outlined in Section 8, and Section 9 outlines some deficiencies in the model. Section 10 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 homogeneous network of workstations with distributed memory[1]. 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 4), 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

[1] This is a temporary restriction due to the different representation of scalar and structures between different architecture and/or operating systems.

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[2] that specifies the number of elements to be passed (unnecessary in sequential C). Unfortunately, this restriction

---

[2] Enterprise expects the caller to allocate sufficient space for the number of values passed or returned. The size parameter is considered a scalar. As such, it is not updated upon return of the asset call.

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[3], 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( int * data, int 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* finishes executing, it copies 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,

[3]An important point about OUT_PARAM variables is that the caller must allocate sufficient space for the OUT parameter. The callee return values will be copied into the caller’s address space.

INOUT causes data to be transmitted 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 4, 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 minimum and maximum 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 *sum += a[i],* 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 *sum += a[i],* 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[4]. 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( int lowerbound, int 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:

[4] The current implementation of Enterprise has the asset waiting for the other assets to reply to the outstanding work before returning its value. Later implementations will address the problem of interrupting arbitrary user code.

```
int AlphaBeta( int lowerbound, int 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[branch];
      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 case, 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.

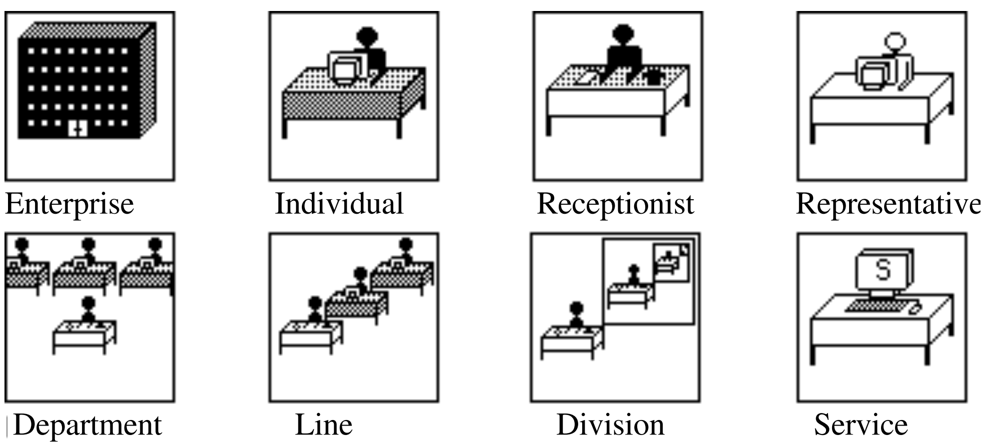


Figure 4.1 Enterprise asset icons.

Each asset represents a commonly-used parallelization technique. 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).

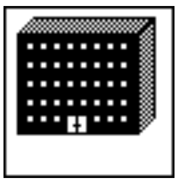
(2) Developers can replicate assets so that more than one process can simultaneously execute the code for the same asset. The interface allows the programmer to specify a minimum and

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, enterprises and the child asset of an enterprise asset.

(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. Simple Assets

Simple assets are building blocks for any Enterprise program. There are three simple assets — individual, representative, and service. They have slightly different semantics which reflects their functionality.

Each asset has code associated with it. The code includes a routine with the same name as the asset; calling this routine is the only way to communicate with this asset. For example, an asset called *Worker* has associated code that includes a routine called *Worker()*. Other assets can make subroutine calls to *Worker* but, because it is an asset, these calls will be invoked in parallel.

4.2.1 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.

4.2.2. Representative

A *representative* is a specialized individual that is used in a division (Section 4.3.4). A representative should contain a call to itself. The representative is where the recursive nature of the algorithm switches from parallel recursion to sequential recursion. The receptionist is considered the leaf node of a division. The receptionist and representative asset in a division share the same code. A representative can only be coerced into a division.

4.2.3. 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. A clock on a wall is an example of a service. Anyone can query it to find the time, and the order of access is not important. A service cannot contain any calls to other assets, but any non-service asset can call it. That is, the asset is fully connected to all other assets.

With service assets, it is possible to simulate shared memory. The following code corresponds to part of Figure 4.5. The service asset, *SharedMem,* accepts two types of calls: one to set the value in shared memory and the other to retrieve the value. Any asset can call *SharedMem* to set/get the value. Of course, *SharedMem 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 */
   SharedMem( SET, &value,  IN_PARAM( 1 ) );
   ...
   /* Get the value */
   SharedMem( GET, &value, OUT_PARAM( 1 ) );
   ...
}
void SharedMem( int access, usertype *value, int size )
{
   static usertype SharedValue;

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

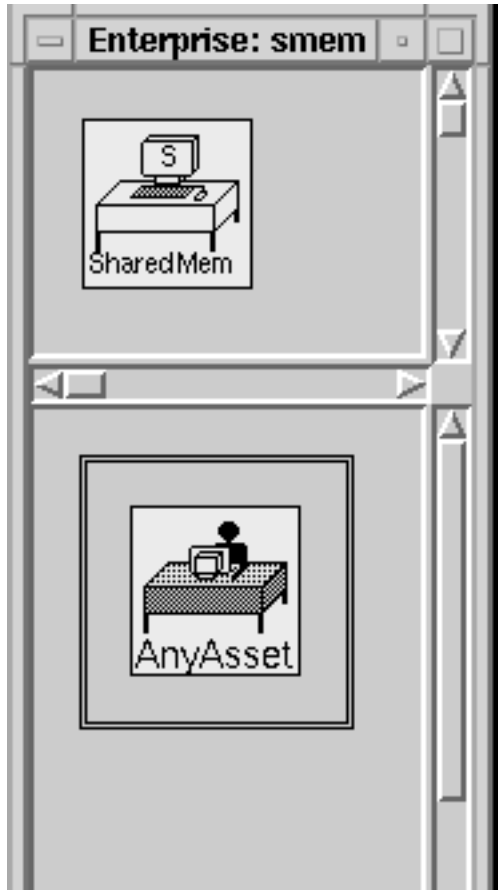


Figure 4.5 Simulating shared memory with a service.

4.3 Composite Assets

Composite assets are assets that contain other assets — parallel templates. The user coerces individual assets to composite assets. The code that was associated with the individual is now placed in the receptionist asset which forms the entry point into and out of the composite asset. The composite asset consists of one receptionist and one or more child assets.

Depending on what type of parallelism is desired, the appropriate asset is selected. To change to another composite asset, the steps involved are: First, coerce the composite asset to an individual and then to the desired asset.

4.3.1. Receptionist

A *receptionist* is the entry point to a composite asset such as a line, department, or division. The name of the receptionist corresponds to the name of the associated composite asset. This specialized individual asset contains user's code and should contain calls to the various child assets. A receptionist receives messages from assets up the hierarchical chain and sends messages down the chain.

4.3.2. 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. If the line is an imbedded asset (inside a composite asset), only the leaf (last) node in a line calls out to the next asset .

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

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

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

void Render( int frame, polygontype polygons, int numbpolygons )
{
      /* 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.

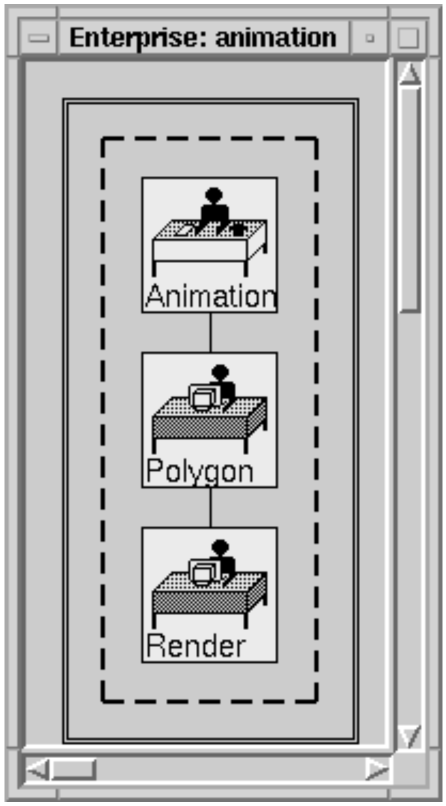
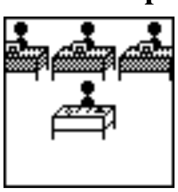


Figure 4.2 *Animation* program as a line.

4.3.3. 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 Linear()
{
   matrix * m;
   int property, dimension;
   ...
   for( ; ; )
   {
      dimension = GetMatrix( m );
      if( dimension == 0 )
         break;
      property = MatrixProperty( m, dimension );
      switch( property )
      {
          case SPARSE:
            Sparse( m, dimension );
            break;
          case TRIDIAGONAL:
            Diagonal( m, dimension );
            break;
          default:
            Gauss( m, dimension );
            break;
      }
   }
}
```

There are three Enterprise solutions to this problem:

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

(2) *Linear* can be transformed into a department asset, containing the individuals *Sparse*, *Diagonal* and *Gauss* (see Figure 4.3). *Linear* 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 *Sparse*, it will execute in parallel with *Linear*. Since these 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 *Gauss* is called, it executes concurrently with *Sparse* and *Linear*. However, if in a subsequent iteration of the loop another call is made to *Sparse*, the second call waits until the outstanding call has completed.

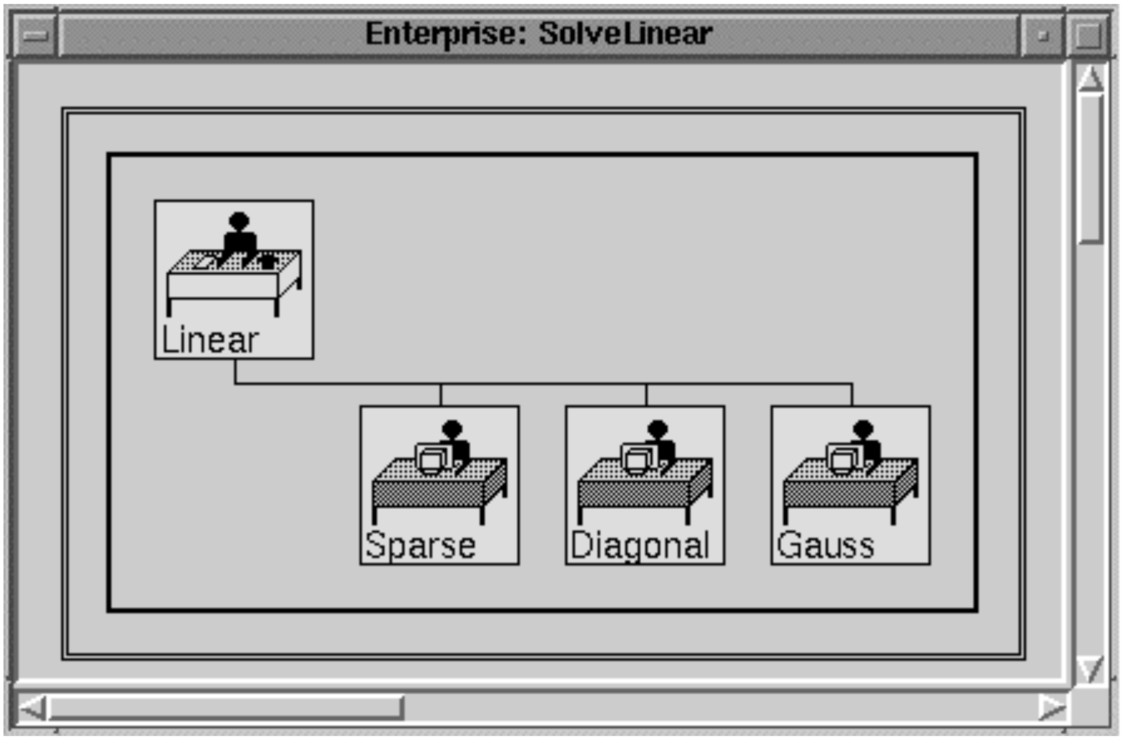


Figure 4.3 *Linear* as a department of individuals.

(3) To reduce the time spent waiting, we can replicate *Sparse*, *Diagonal* and/or *Gauss* 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 matrix solving routines, such as:

```
Diagonal( 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 matrix solving routines could be modified so that the last parameter is not needed.

It requires only a simple code change to convert the sequential *Linear* 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.

4.3.4. 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( int * a, int size )
{
   int pivot;
   if ( size > threshold ) {
      pivot_index = partition(a, size);
      QuickSort(&a[0], pivot_index);
      QuickSort(&a[pivot_index+1], size - pivot_index);
   } else {
      SequentialSort(a, size);
   }
}
```

For this program, the division should be defined with a breadth of two (since the code contains two recursive calls to *QuickSort*) 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. The 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. Adding another level to the division in Figure 4.4 means that there would be two levels of parallel recursion before switching to sequential execution. 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.

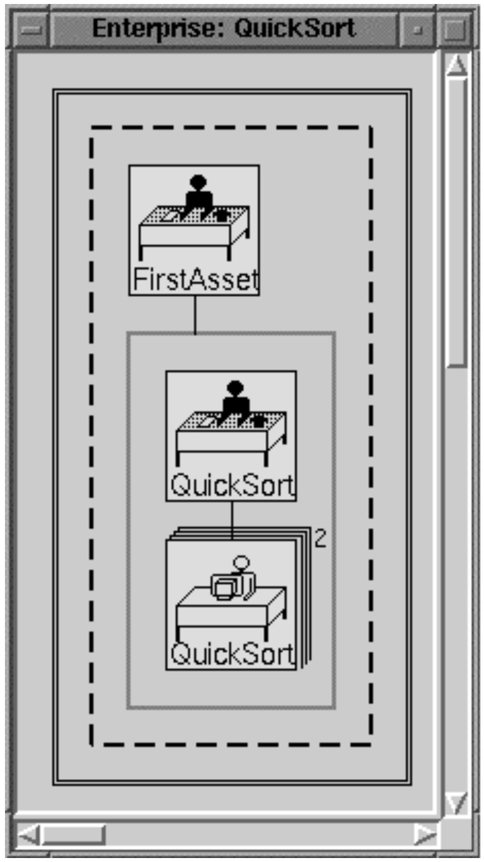


Figure 4.4 *QuickSort* using a division asset.

5. The User Interface

This section contains an 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 in the tree. The parallel version uses the principal variation splitting algorithm, which recursively descends the leftmost branch of the tree, searches the siblings of the leftmost branch in parallel, and backing up the minimax result to the root of the tree.

The complete source for AlphaBeta can be found in the Enterprise release in the subdirectory Examples/AlphaBeta. 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 there are searches to be performed. *Draw* is used to graphically display the progress of the search.

```
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 *width-1* 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 */
   width = 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 <= width; 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 <= width; 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 uses the negascout enhancement. First, it searches the first move with the full (*alpha*, *beta*) window, and then, the 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 */
   width = Generate( branches );

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

```
        if( result > alpha && result < beta )
           /* Result is outside the window; re-search to get the
              correct value. */
           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 tree. A separate program, *Pixel*, is used to avoid conflicts between X windows event loops and communications kernel message passing. (The code and associated makefile for *Pixel* is found in the subdirectory, Examples/AlphaBeta/Draw in the distribution.)

```
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 to *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* or *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, there either a stand-alone image created for all users or a single-user image and changes file that a user must copy to their home directory. Check with the installer or read the appropriate file in the distribution tar file. To run Enterprise using the stand-alone image, the user starts a Smalltalk interpreter using the stand-alone enterprise image (enterprise.im). If the single user image is used, 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. In either case, the user interface will then appear as shown in Figure 5.1.

Before we proceed with this section, some conventions used in this document need explanation. When the mouse pointer is inside the scope of a Smalltalk window, a context-sensitive menu is popped up by holding down the middle mouse button (the <operate> button). Moving the mouse so that it points to different items or areas in the window results in different menus being popped up. To select a menu item, move the mouse up and down while continuing to press the middle button. The right mouse button will popup a Smalltalk system menu. This document will not discuss Smalltalk system menus. To select an item, say in a list, use the left mouse button (the <select> button) to highlight the item. This is true as well when selecting a button in a dialog display. We use a different typeface, This is an example, to indicate menu or items (buttons or list views) in a window or dialog view.

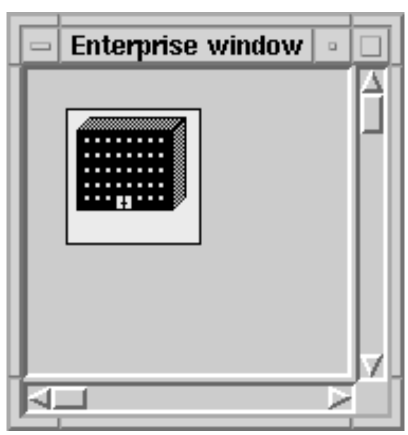


Figure 5.1 A new Enterprise program.

Enterprise provides several views of an application. The Design View is used for drawing the asset diagram, editing, compiling and running a program. The Animation View (discussed in Section 5.9) allows the user to view a completed program execution as an animation of asset calls and returns. The Replay View (Section 6) allows the user to deterministically replay an execution run and invoke some debugging tools.

This Design View initially contains a new enterprise asset. The user has two options available in the menu[5] , Open and Quit. Selecting Open either creates a new program or opens an existing Enterprise program. We wish to create a new program — select Open.

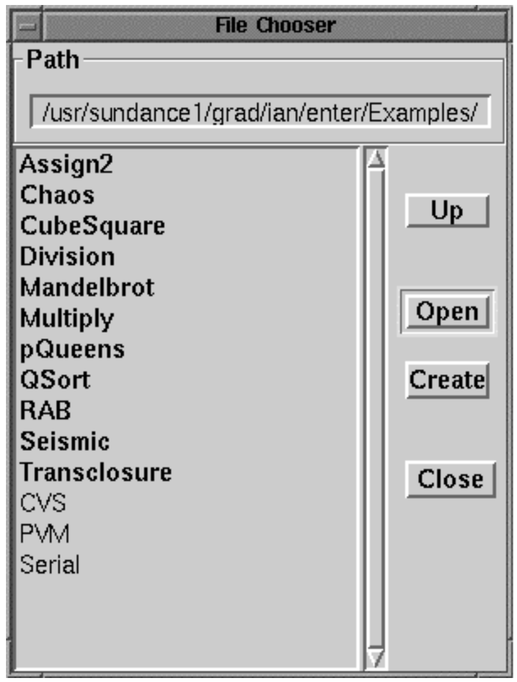


Figure 5.2 The program selection dialog.

Figure 5.2 shows the Open window. Enterprise programs are shown in bold face while other directories are shown in plain text. To move up or down in the directory tree select the Up or Open buttons. A short-cut is available by directly typing in the Path text field. To create a new program in the current directory, press the Create button. A dialog box opens for you to type in the new program name. The resulting Enterprise program directory will be created in the current directory defined by the Path text field. Typing the name *AlphaBeta* into the dialog box and pressing <ENTER > names the program, with the result shown in Figure 5.3.

When a new Enterprise program is created, a directory structure is automatically set up to organize the files associated with the application. More details can be found in Section 7.1. For an application, such as *AlphaBeta*, the following directories are maintained by Enterprise:

---

[5] There is a third option in the single-user image (Make standalone) which creates the stand-alone image. This is not discussed in this document.

Graph Various files for the compiling, running, and recording events of Enterprise binaries are kept here.

Assets Source code for all the assets (using a ".e" suffix on the file names).

Src The Enterprise precompiler preprocesses ".e" files to produce ".c" files that are placed in this directory.

User Enterprise puts other user-application code (such as support routines) here.

Obj The Src/*.c and User/*.c files are compiled and placed in Obj.

Include Enterprise puts all include files (".h") in this directory.

Tmp Used for temporary Enterprise files.

Bin Each architectural binary produced by Enterprise is placed in Bin. Only one binary per architecture is produced.

Data This is a convenient location of data files. This directory is intended to be a consistent logical location regardless of which processor the binary is running on.

Out Contains the standard output (*stdout*) from a program execution for all the assets.

Err Contains the standard error (*stderr*) from a program execution for all the assets.

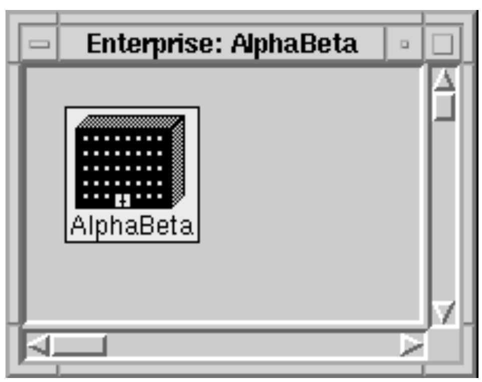


Figure 5.3 A program named *AlphaBeta*

When the mouse is pointing on the *AlphaBeta* asset icon and the middle mouse button is pressed an asset-specific menu will appear. In this case, the menu choices Name, Expand, and Expand fully are available to the user. Selecting Expand from the menu will expand the enterprise asset to reveal the single individual it contains, giving the view shown in Figure 5.4. Note that the individual has been given a default name of *asset1*. If the asset contains a more complicated structure, Expand fully will expand all assets in the hierarchy.

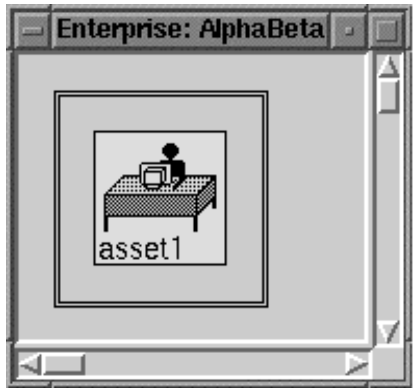


Figure 5.4 An enterprise containing one individual.

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 5.5. 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).

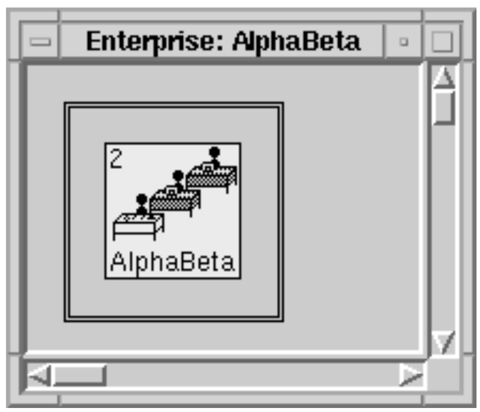


Figure 5.5 A program containing a line.

The line can be expanded by selecting Expand from its asset menu. Doing this reveals a receptionist named *AlphaBeta* and an individual named *asset2* as shown in Figure 5.6.

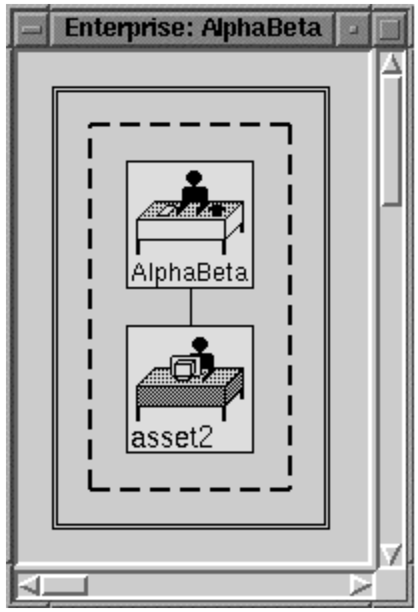


Figure 5.6 An expanded line asset.

As discussed earlier in this section, menus are context-sensitive. Here is an example of this. The double line represents the enterprise, the dashed line represents the line, and the icons represent the receptionist and individual. Clicking inside the double-line enterprise rectangle but

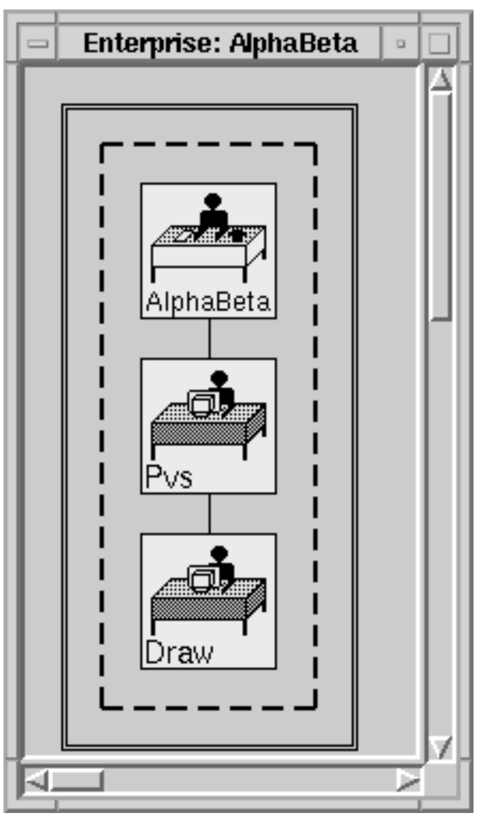


Figure 5.7 A three asset line.

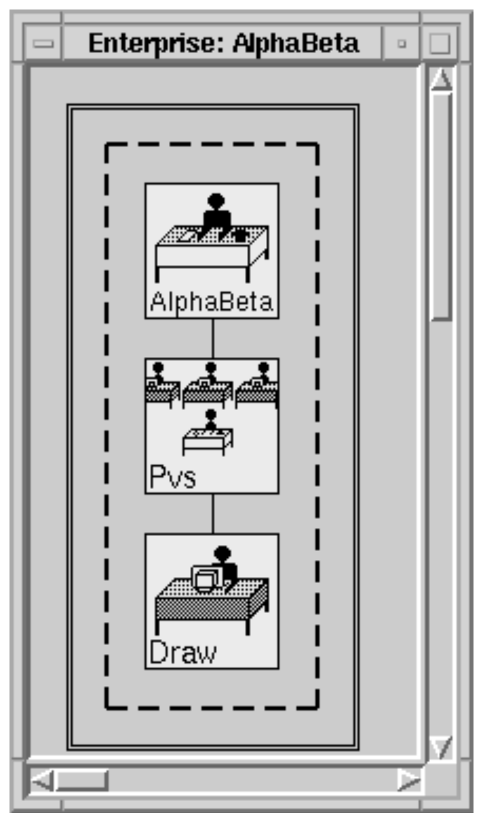


Figure 5.8 A line asset containing a department asset.

outside the dashed-line rectangle will display the asset menu for the enterprise asset. Clicking inside the dashed-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. Clicking outside the double-line enterprise rectangle will display the Design View menu.

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 5.7.

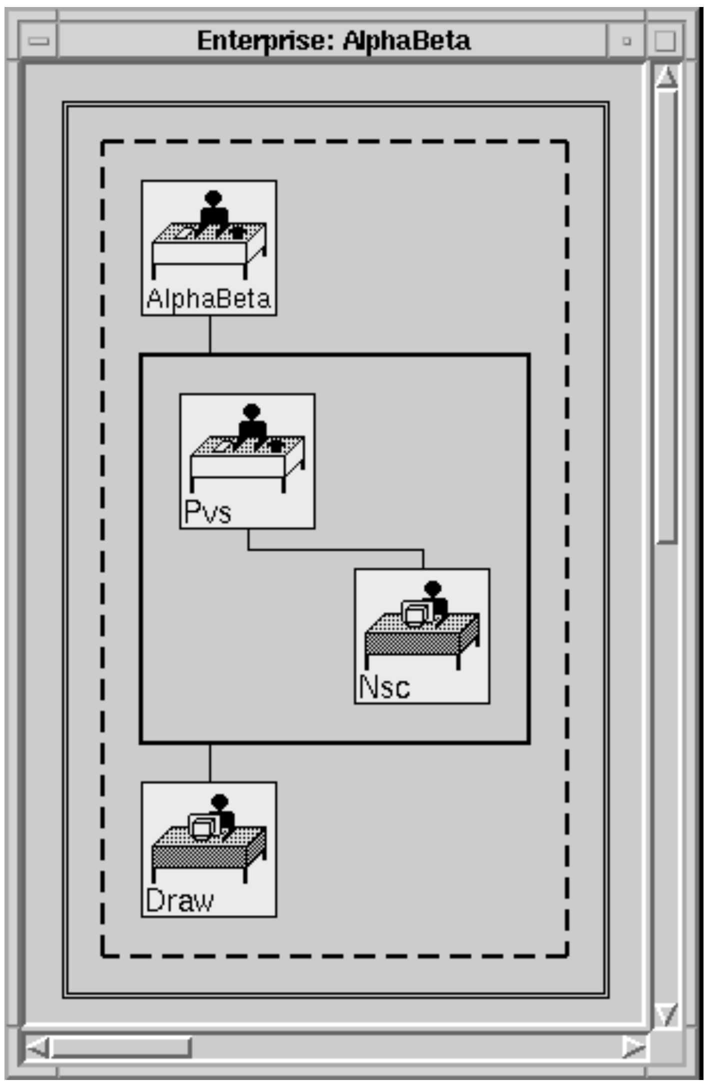


Fig 5.9 Asset graph for *AlphaBeta*.

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 5.8. 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 5.9.

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 can contain many subroutines, but it must contain a subroutine with the same name as the asset. Procedures that are common to several assets should 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.

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

There are several options that can be set by the user. If they are set in the Design View, they apply to the entire program. If set at the asset level, they apply only to that asset and any other assets lower down in its hierarchy (children) unless specifically overridden. What is usually done is to have the options set globally (see, for example, compilation in Section 5.6).

A dialog box for setting compilation and runtime options of assets can be opened by selecting options from an asset's menu. Figure 5.10 shows an asset options dialog box. The Debug or Optimize toggle buttons and the CFLAGS text field specify options for compiling an asset[6].

---

[6] Typically, asset-specific defines or include directives are placed here. Some compiler flags (-o, -O, -g, and -c) are not allowed. Enterprise automatically fills these in at compile time.

The CFLAGS inherit their values from assets up the graph (parents) unless explicitly overridden. The user is given a chance to continue the inheritance when pressing the Accept button when something is present in the CFLAGS field. Otherwise, inheritance is defeated until the user puts in an empty string.

If Output Windows or Error Windows toggle buttons are set to *on*, then windows will be opened at runtime to display standard out (*stdout*), and standard error (*stderr*), respectively. The Include list view is used to specify a list of machines which this asset must run on. The Exclude list view specifies a list of machines which this asset must not run on. The list of active machines, Available Machines, contains the list of machines[7] that can be selected for either the Include or Exclude lists. To add, delete, or modify the active machine list, the user must open the Run dialog box (see Section 5.7.1).

To add a machine to either the Include or Exclude list view, move the mouse pointer into Available Machines list view and select one or more machines. Then, press the middle mouse button to popup a menu. This menu contains two menu items — Include and Exclude. Selecting Include will move the selected machines from the Available Machines to the Include list. Similarly, the Exclude menu option moves the selected machines to the Exclude list. To remove machines from either the Include or Exclude list, select the machine name(s) and then press the middle mouse button. The resultant popup menu has one choice, Remove. Selecting this menu option moves the chosen machine(s) from the current list view to the Available Machines list.

The Accept button commits the current settings to this asset and closes the dialog view. The Revert button restores the last saved values, while the Close button closes the dialog box without saving the current setting.

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 the Asset Options dialog box for *AlphaBeta* shown in Figure 5.10, the Output Windows radio button was set to *on* by clicking with the left mouse button. This will cause an output window to appear at execution time for *AlphaBeta* and all assets that are contained in *AlphaBeta*. When the desired options are selected, press the Accept button to accept and remember the options.

[7]The Available Machines list is created by taking the current active machine list (see running an Enterprise program, Section 5.7.1) and removing any machine names in either the Include or Exclude list views for this particular asset.

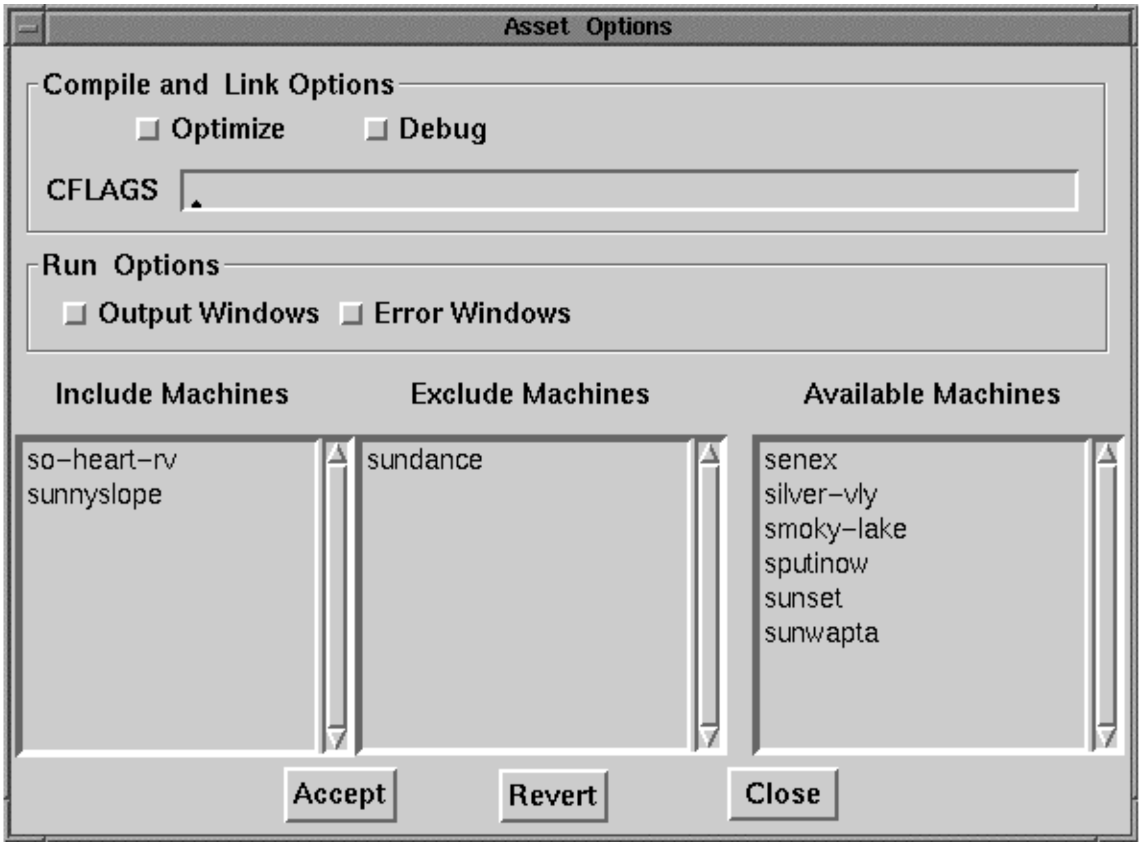


Figure 5.10 Asset options dialog box.

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[8]. Only one compilation can be active at a time.

The compile dialog box consists of several fields:

- A scrollable text field which will contain the output from a compilation, including any error messages and warnings,

[8] For this version of Enterprise, this is true only if the asset or user's source code has been modified. If any include files are modified, the dependent source files will not be updated!

- A toggle switch, Verbose, which determines how much compilation output should be displayed,
- Toggle switches, Debug and Optimize, to specify the type of executable desired,
- A CFLAGS text field to specify global compiler flags[9],
- A Libraries text field which supplies the list of user libraries and their locations for the program,
- The Compile button to initiate a compilation of the program,
- The Clean button "cleans" up the account by removing any temporary Enterprise files as well as all object and executable files,
- The Abort button stops the compilation that is in progress,
- The Revert button allows the user to recover the stored preset values for all the fields in this window, and
- The Close button closes this window.

Figure 5.11 shows 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.

---

[9] Typically, asset-specific defines or include directives are placed here. Some compiler flags (-o, -O, -g, and -c) are not allowed. Enterprise automatically fills these in at compile time.

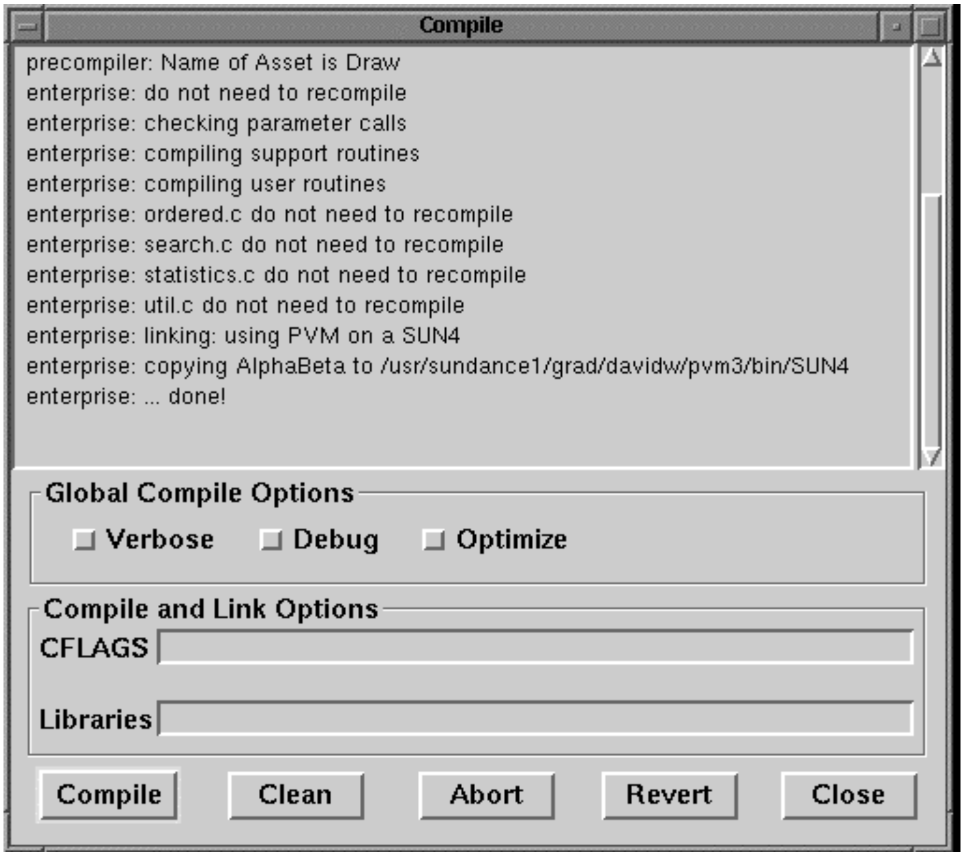


Figure 5.11 A compile dialog box.

5.7. Running the Program

Enterprise programs can be run by selecting Run from the Design View menu. A run dialog box consists of several fields:

- A scrollable text field for displaying runtime output,
- A scrollable list for displaying the names of active or inactive machines to use while running,
- A toggle switch labeled Create event log for enabling logging of events (used for performance monitoring and execution replay, see Section 5.9),
- A toggle switch labeled Run sequentially for executing the program sequentially (parallel asset calls become sequential subroutine calls),
- A toggle switch labeled Debug Mode that causes all future accesses and asset calls to be printed to the standard output for the process,
- Text fields labeled Input file and Output file for specifying input and output files,
- A text field labeled Args for specifying runtime command line arguments,
- A Run button for starting a run,

- An Abort button for aborting a running program,
- A Revert button for recalling the previous run dialog settings, and
- A Close button for closing the run dialog.

A run dialog box is shown in Figure 5.12. In this window, event logging is enabled and the program will be run with command line arguments of "*Data/in*". For the *AlphaBeta* program, the argument is the name of the input file.

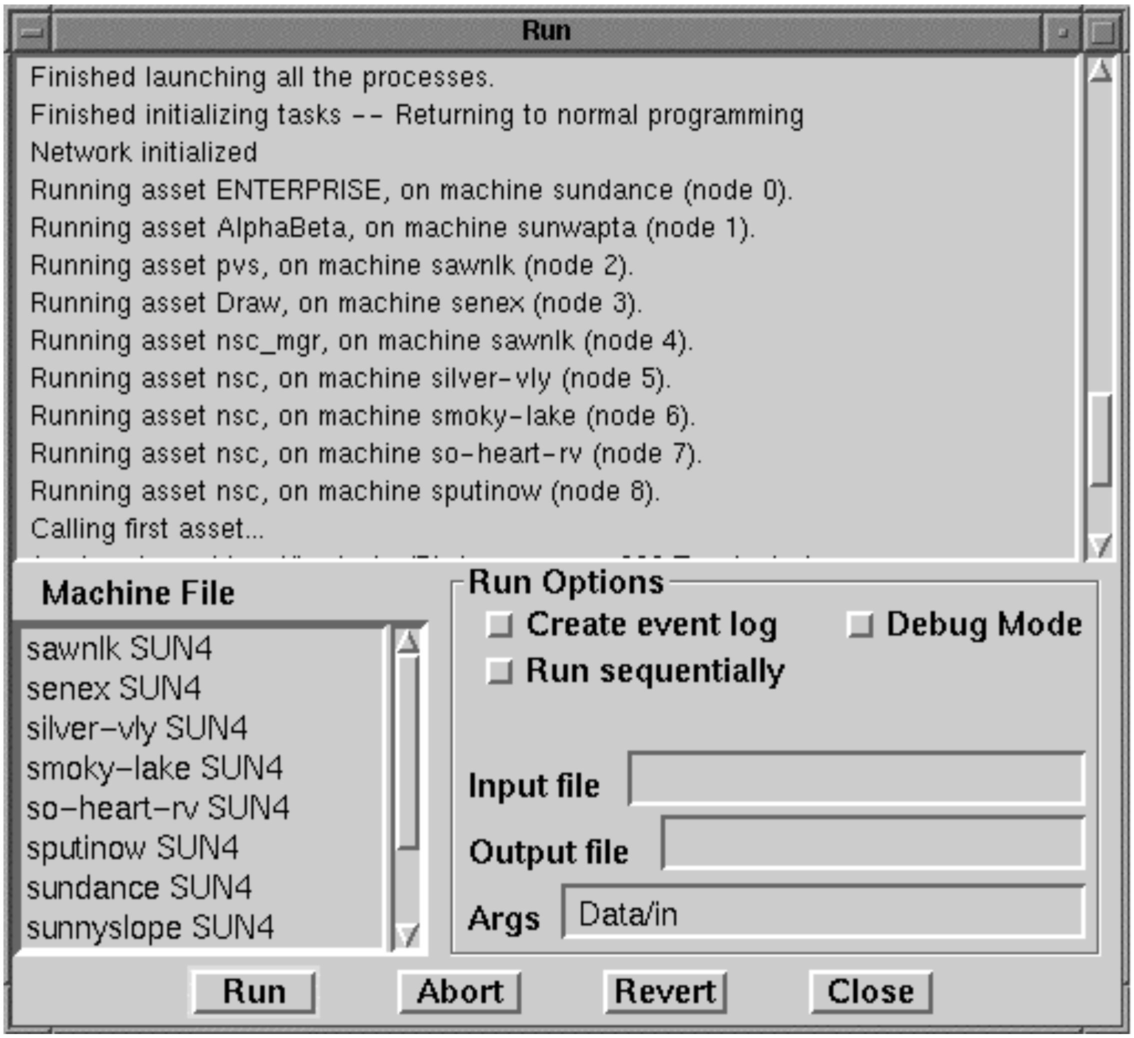


Figure 5.12 A run dialog box.

5.7.1 Machine List

Before the compiled program can be executed, the system must be told which machines it can use. The list of available machines is shown in the scrolling list in the bottom left hand corner of the dialog. Machine names with a number character (#) in front of them are not used during a run (inactive). The other machines are considered to be the active machine list. To modify the list of machines, click the middle mouse button inside of the scrolling list to get the machine list editor menu.

To include an excluded machine, select a machine name with a # in front of it using the left mouse button, then choose Include from the machine list editor menu. The # should disappear from the front of the machine name.

To exclude a machine already in the machine list from the list of available machines to run on, select a machine name with the left mouse button, then choose Exclude from the machine list editor menu. A # should appear in front of the machine name.

To add a new machine to the list of machines, select Add from the Machine File list editor menu. The Add Machine dialog will pop-up as shown in Figure 5.13. The dialog has a text field for the name of the machine, and a series of radio buttons to select the type of machine. Only the valid architectures supported by Enterprise are displayed[10]. Click the Add button in the dialog to add the values in the text field to the machine list. Click Close to dismiss the dialog.

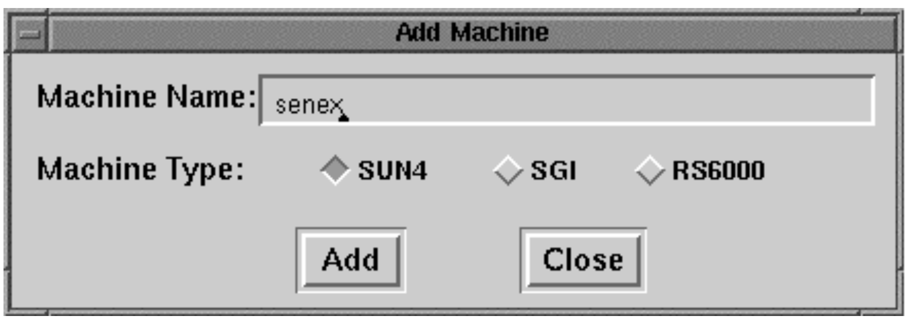


Figure 5.13 An Add Machine dialog.

To remove a machine from the list of machines, select the machine name with the left mouse button, then choose Remove from the machine list editor menu.

To save the values in the machine list choose Save in the Machine list editor menu. This has the side effect of modifying the Include and Exclude options in the Asset Option view. Each asset is checked to see if the Include or Exclude lists contain inactive machines. If so, the inactive machine names are removed from the Include or Exclude list.

To revert the values in the Machine File list editor to the previously saved values, choose Revert in the machine list editor menu. When the program is run, the current list of machines is saved.

[10] Currently, only the Sun OS 4.1.3 (SUN4), IBM AIX (RS6000), and Silicon Graphics Release 5 (SGI) operating systems are supported.

5.7.2 Running an Enterprise 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. After the run, the windows are left open until explicitly closed by the user. Note that on some systems, the program output displayed in these windows may lag behind the execution.

Figure 5.15 shows a sample output from running *AlphaBeta*. Included is a window created by the *Draw* routine showing the search tree built by the program.

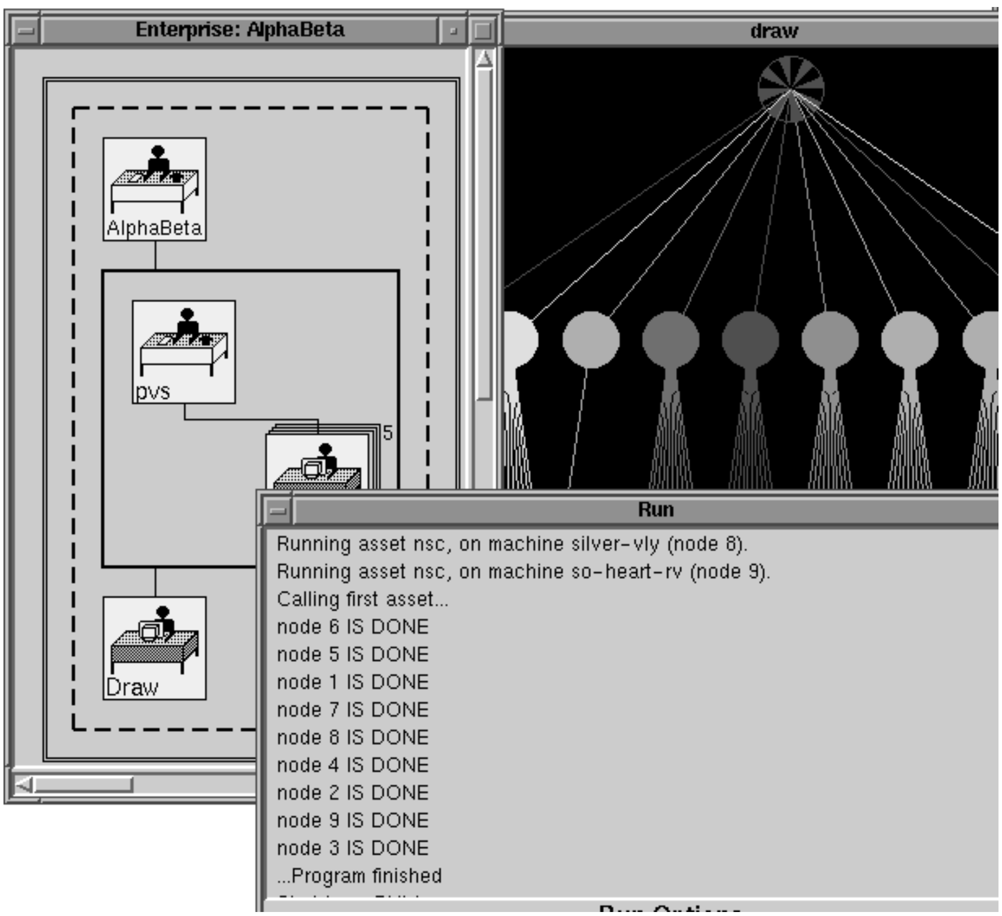


Figure 5.15 Running the *AlphaBeta* program

5.8. Specifying Logged Parameters

The Design View contains a facility for specifying which parameters of an asset call are to be logged during program execution. This information is used in the Animation and Replay Views to allow a program execution to be recreated later. These tools are useful for debugging and performance tuning.

When a program is run with event logging enabled, an event file is created. Enterprise knows the number and type of all asset parameters and return values. Each is given a default logging specification. The user can increase or decrease the amount of information gathered.

The default specification is to log all scalars[11] on asset calls and scalar return values when the asset returns. All vectors of scalars also have their first three and last two values logged on an asset call. The first asset in a program, however, has no logged parameters. The current version of *Enterprise* has, as yet, no facility for logging structures or pointers to data types other than scalars. This deficiency will be addressed in the future.

The user can modify the default settings for parameter logging by selecting Parameter Logging from the pop-up menu for an asset while in the Design View. Such customization can be done only after the asset has been compiled. The interface then displays a dialog box containing a list view of the parameters and their logging specifications, together with a specification editing facility. Figure 5.16 illustrates a Parameter Logging Editor for an asset named `Square`, which has two parameters (`i` and `y`) and a return value (`^`). The asset is identified by name in a read-only field at the top of the dialog box. The Logging Specification list view on the left provides the current logging specification for all the parameters and the return value. Some parameters may appear in two lines, one for input logging and another for output. When an entry is selected, pressing the *'Operate'* button in the list view pops up a menu for changing the logging specification for the highlighted selection. This context-sensitive menu only shows the relevant menu choices. The possible choices are:

Log on input
Log on output
Do not log

---

[11] *Scalars* refers to the fundamental built-in C types: char and the int types, signed and unsigned, and the floating-point types.

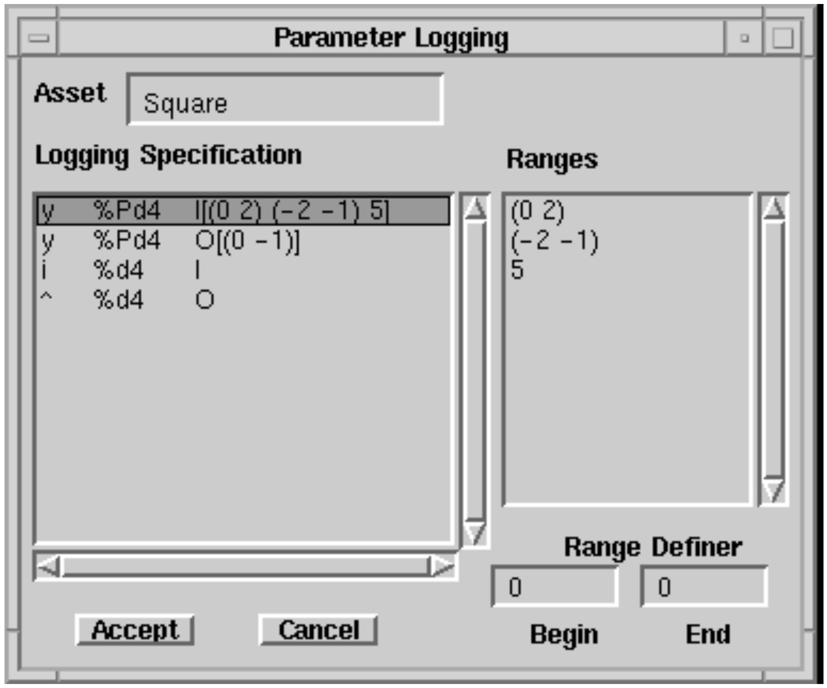


Figure 5.16. Parameter logging editor

These menu choices change the *log code* for the parameter. The log code, appended to the parameter entry, indicates if logging is performed on the parameter when the asset is called (**I**, for ***i**n*), when the asset returns (**O**, for ***o**ut*), or not at all (**X**, for *e**x**clude*). If a parameter is logged on both *in* and *out*, then the parameter must be given two separate specifications. This situation does not occur with scalars but can occur with arrays. Scalar parameters are by definition input parameters only, so they can only be logged on input. Return values (`^`) can naturally only be logged on output. Each parameter entry consists of the parameter name, followed by its type, size in bytes, and a description of logged ranges for arrays. Scalars are designated by a %d (integer), %f(float), or %c (char). Arrays of scalars, designated by a `%P` followed by a `d`, `f`, or `c`, can be logged on input and/or output. As mentioned earlier, more complex structures, identified as %R, cannot currently be logged. A collection of ranges is shown in square brackets [ ] for arrays of scalars being logged. Each range is either a single integer or a pair of integers separated by a space and enclosed in parentheses ( ). Following the C convention, 0 refers to the first element in an array. Negative integers denote elements at the end of an array. Thus, -1 refers to the last element of an array. The range (0 -1) denotes that the entire array is logged. For readability, if no range is logged, it is displayed with ellipses as **[. . .]**.

When an array of scalars is selected in the Logging Specification list, as in Figure 5.16, the Ranges list view on the right becomes active and displays the ranges in a list form. The Range Definer fields in the lower right can then be used to specify a Begin and End point for a new logging range. If the Begin and End points are the same, or if only a Begin point is specified, a single element will be added to the Ranges list. A context-sensitive pop-up menu can be accessed in the Ranges list. The possible selections are:

Add range
Delete range

Add range performs the function of adding the range specified in the Range Definer to the Ranges list. A notifier appears if the range is illegal or not yet specified. After a range has been added, the Range Definer resets to **0 0**. The added range appears at the end of the list if no entry in the Range list was selected. If an entry was selected, the new range will be added immediately above the selection, in accord with Smalltalk conventions. Delete range removes the selected range from the list. The Logging Specification list automatically updates when changes occur in the Ranges list.

Figure 5.16 shows a parameter *y*, which is an array of integers (`%Pd4`), each consisting of 4 bytes. On input, the first three elements (indexed 0, 1, and 2), the sixth element (indexed 5), and the last two elements (indexed -2 and -1) are to be logged. On output, the entire array (indexed from 0 to -1) is to be logged.

When logging specification changes have been made, the Accept button, initially grayed out, becomes active. When the Accept button is clicked, the new specification is saved and the dialog box closes. If the Cancel button is clicked, the dialog box closes without modifying the original logging specification.

It should be stressed that the probe effect of event logging is increased by the additional logging requirements specified for parameters. Keeping this fact in mind, the user should avoid logging unnecessary information. On the other hand, reducing the logging information reduces the control potential of the debugging facilities (Section 6) and requires the user to rely on sequential debuggers for examining parameter information. There is no additional probe effect during execution replay, regardless of the amount of breakpoint analysis being performed, since the replay is fully deterministic.

5.9. 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. The Animation View allows us to see if there are any performance bottlenecks and observe if the program is making full use of its resources.

The Animation View is obtained by selecting Animate from the Design View menu. After changing views, the program appears as in Figure 5.17. The animation view is similar to the

Design View in Figure 5.9, 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.

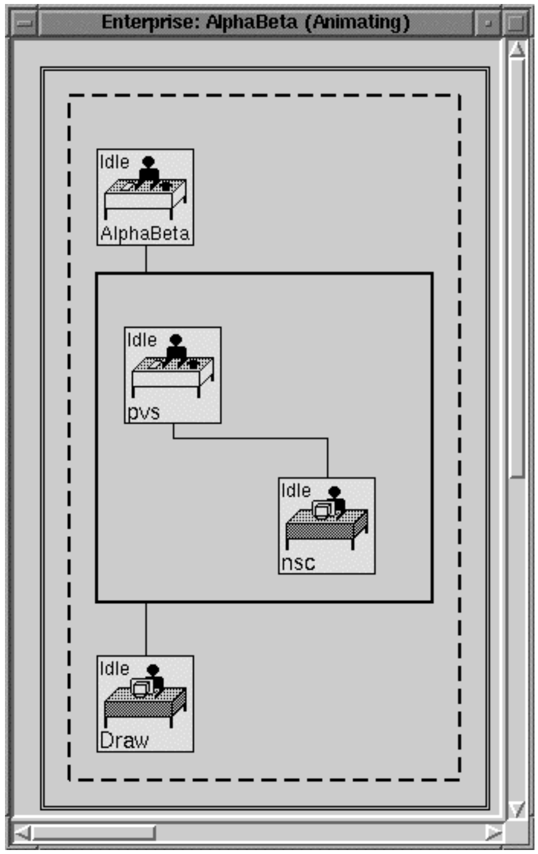


Figure 5.17 The animation view of the sample program.

The menus that appear are different as well. Clicking in the Animation View, outside of the assets displays the main animation menu.

The choices are:

Start
Step
Set options
Design view

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 5.18 To stop the animation, select Stop from the Animation View menu while the animation is running. Depending on the animation options selected, this may take some time. During an animation, selecting the Stop menu causes the animation menu to change. Start is removed with Reset and Resume added. Selecting Reset ends the animation and the menu reverts back to the original Animation View. Selecting Resume continues the animation.

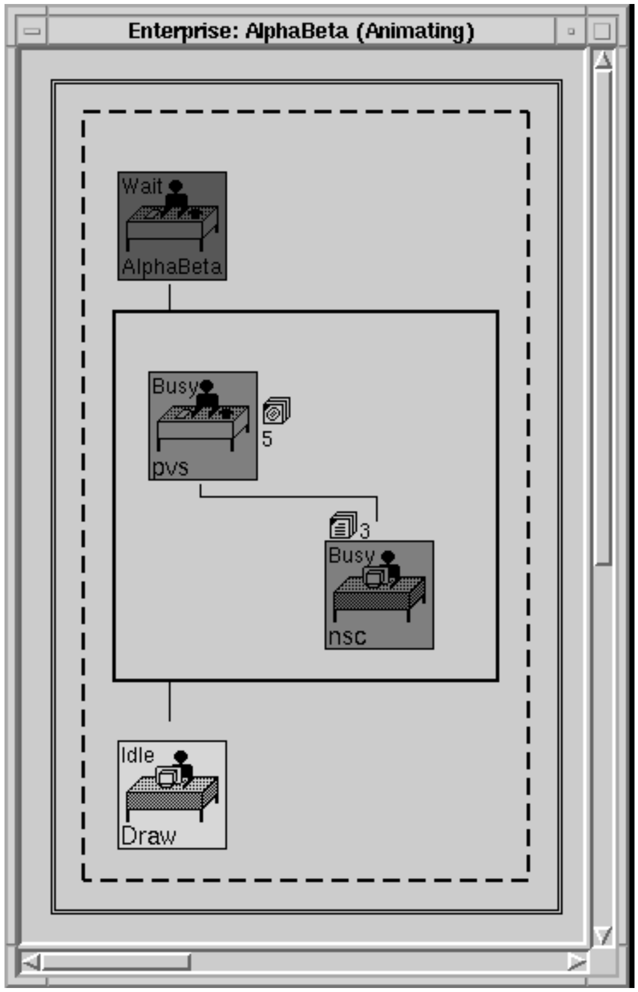


Figure 5.18 A point in the animation.

Step lets the user single step through the animation through one event. Design view returns the user back to the Design View. Set options opens a dialog box with five parameters the user can set to speed up or slow down animation. Figure 5.19 shows the dialog box.

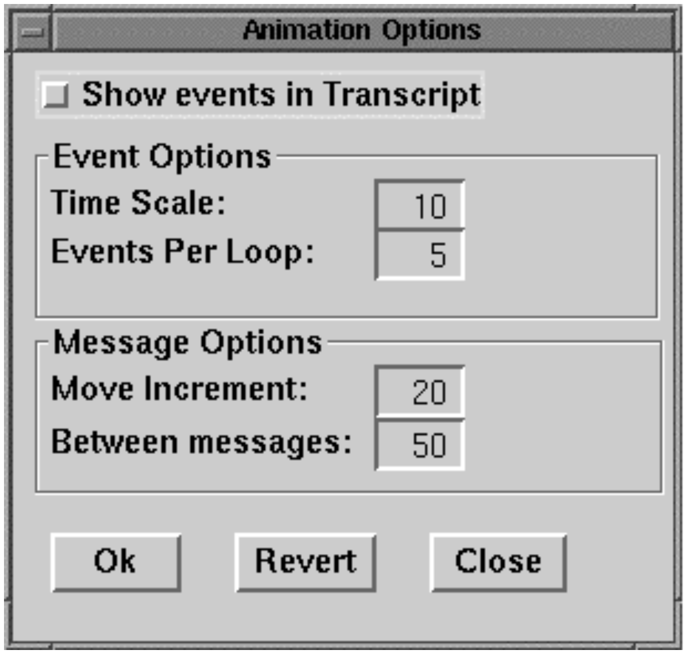


Figure 5.19 Animation options dialog box

The user has three buttons along the bottom of the dialog. Ok stores the current option settings displayed; Revert restores the options to the previously saved values; Close exits this window. The toggle button, Show events in Transcript, which appears only in single-user mode, writes Enterprise logged events as they are animated in the Transcript window.

Time Scale is intended to map the virtual time to the real-time scale. If there is a large time difference between messages, setting this number larger will allow the animation to progress quickly. Setting this value lower will slow down the animation. Currently, this value is not used.

Events Per Loop determines the number of events that are processed for one animation cycle. This cycle cannot be interrupted by the user. Setting the number high is good if an overall view of the program performance is needed. If a more detailed look at the performance is needed, set the number lower. Note, not all Enterprise events are animated. For example, if an assets goes from busy to idle and back to busy in one cycle, this change is not seen.

Move Increment set the number of pixels the message will move along its delivery path. Between messages sets the number of pixels between consecutive messages along a delivery path.

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 5.20. Selecting one of the messages in the queue displays the values of the logged parameters in the C procedure call to *Nsc*.

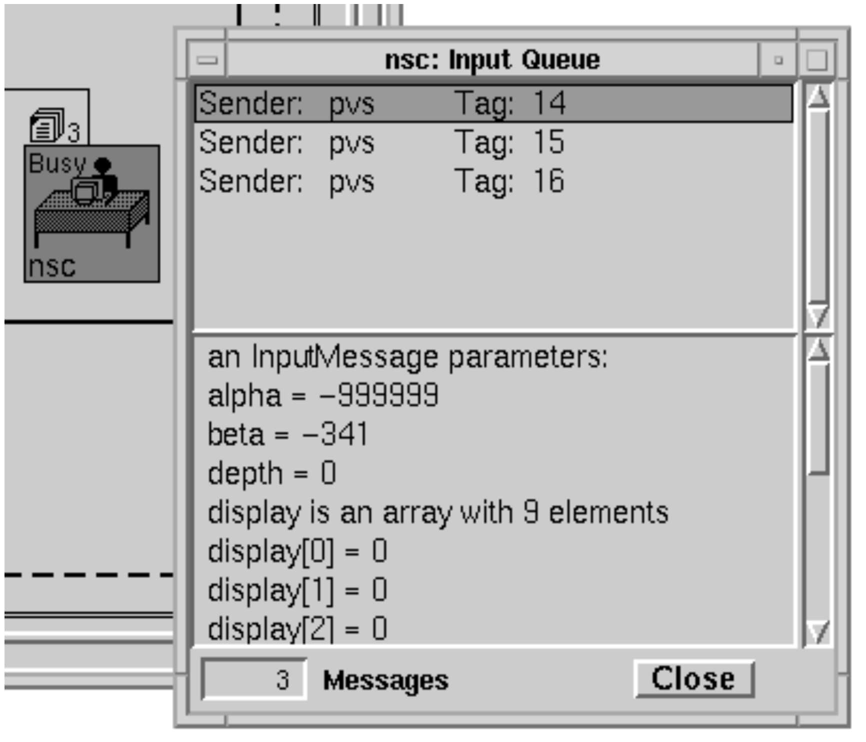


Figure 5.20 A message inspector

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

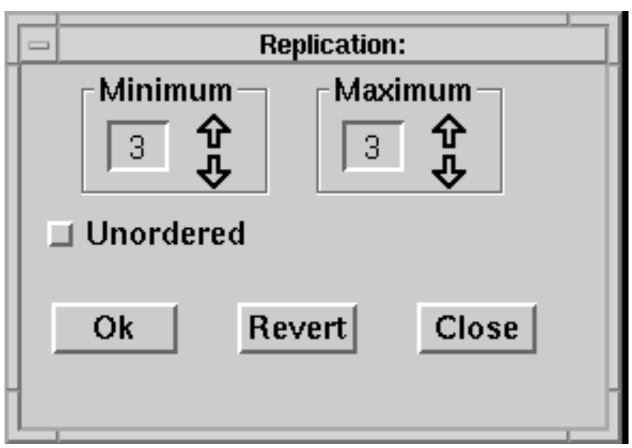


Figure 5.21 The replication dialog box.

are in the Design View, we select Replicate from *Nsc*'s asset menu, causing the replication dialog box to appear as shown in Figure 5.21.

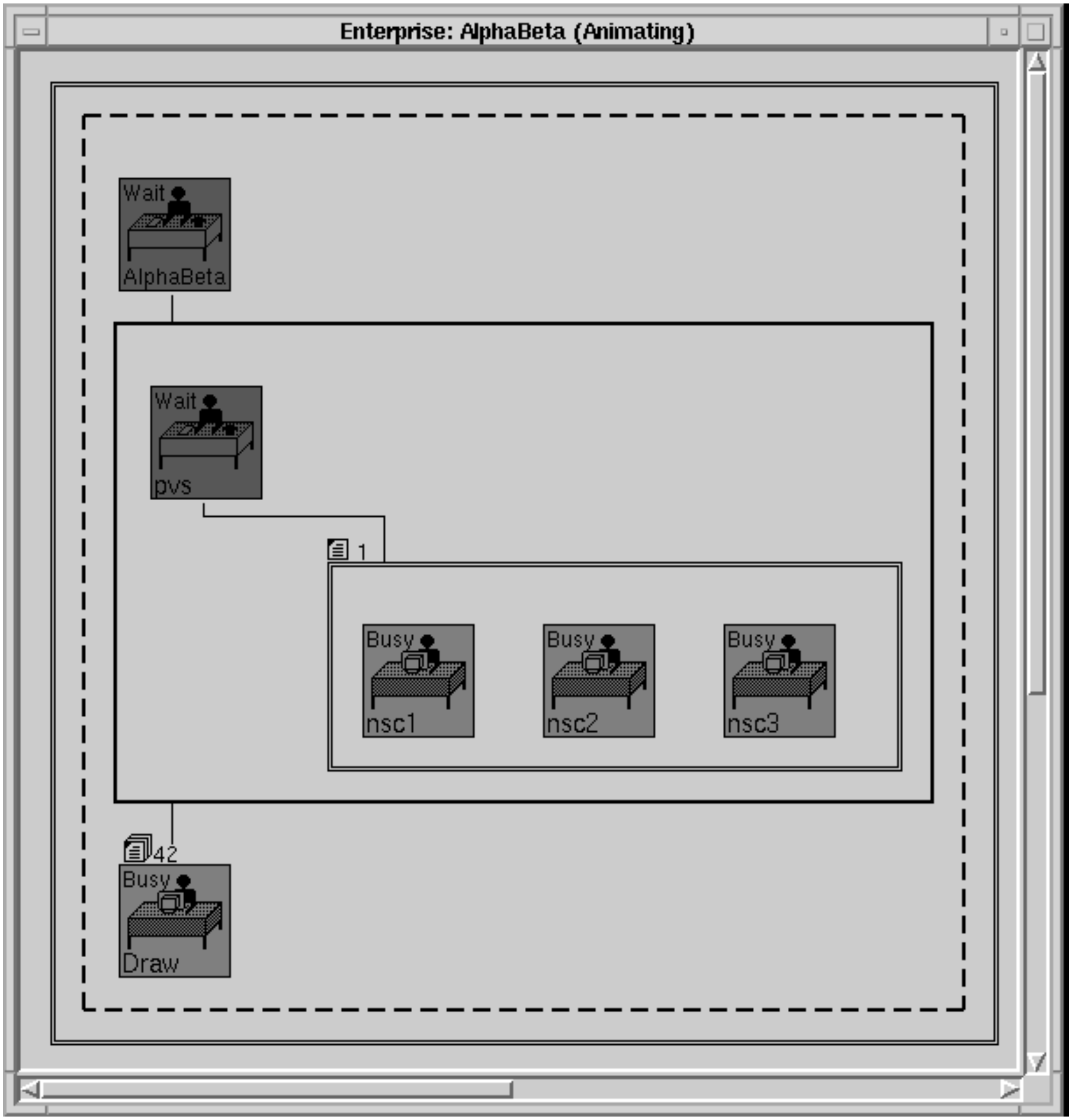


Figure 5.22 Animation view after replicating Nsc

We replicate *Nsc* three times by clicking on the up arrow until the maximum replication factor becomes 3[12]. Now we can close the box, and re-execute the program[13]. When we switch to the

---

12 Only the maximum replication factor is currently used. Later implementations of Enterprise will allow dynamic process replication. That is, start off with, say, one replica and as demand increases, add more replicas to the maximum allowed. In this case, three.

13 The current implementation requires a recompilation of the program only if changing the replication factor from one to replicated or replicated to one. Typically, this is very fast as none of the user's code is recompiled.

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 5.22.

6. Debugging in the Replay View

The Replay View allows controlled replay of a program from an event file. It is entered from the Design View main menu by selecting Replay. The Replay View is nearly identical in appearance to, though different in functionality from, the Animation View. While the Animation View allows for the "simulation" of a program's execution, the Replay View provides the means to actually perform an identical re-execution. Like the Animation View, replicated assets are fully expanded, message passing is animated, and message queues can be monitored.

Although logic errors in a program can sometimes be debugged by post-mortem examination in the Animation View, there are times when it becomes necessary to actually re-execute the program deterministically along the path captured in the event log. This is possible assuming that there are no non-deterministic constructs in the sequential code of the assets themselves. This forced replay facility combined with a message-level breakpoint facility and selective access to standard sequential debuggers all together constitute the *Enterprise* debugger.

6.1. Non-Determinism

Although the opportunities for non-determinacy are fortunately reduced in the *Enterprise* model, the phenomenon still occurs. Within the *Enterprise* model, there are three identified circumstances where critical race conditions can develop. The most common situation involves the Receive Message event. If the receiving asset is connected to only one sender, as in a simple Line or simple Department, the behavior of the receiver is deterministic. The communication manager guarantees a first-in-first-out protocol on the pipe connecting the two assets. Moreover, messages cannot get lost, assuming there is no hardware malfunction or system software error. However, if the receiving asset is connected to more than one sender, as is the asset *Draw* in the *AlphaBeta* program, a race condition can develop. In such a situation, each of the assets in the Department *Pvs* could non-deterministically send messages to *Draw*, potentially impacting the end result of the computation.

Another non-deterministic situation involves a manager sending a message to its replicated assets. A message might go to replica 1 during one execution, but to replica 2 on another. Although all the replicas execute the same code, the use of static variables, which retain their values between calls, could alter the results of a computation depending in the sequence of replica calls.

A third non-deterministic situation occurs when an asset receives a reply from an unordered replicated asset. When an asset is designated as unordered, the futures associated with return values from that asset can be used by the calling asset in any order. This unpredictable ordering could easily produce differing results if the unordered option is used inappropriately.

6.2. Main Menu

The main pop-up menu in the Replay View is activated while the cursor is anywhere outside the program view boundary. The menu choices are dependent upon the current state of the system. The possible choices are as follows:

Start	Selecting *Start* launches a re-execution of the program according to the event file's sequence of events. Upon launching, the Run Box is raised to the foreground, or created if one does not already exist. The initialization messages for the program and any standard output appear in the text collector view of the Run Box.
Stop	This is the single choice available when replay is running. When selected, the replay and animation pause between discrete events.
Resume	Once a replay has begun but has been stopped by a triggered breakpoint, the *Stop* option, or after stepping, the choices *Resume* and *Reset* replace *Start*. *Resume* allows the replay to continue from where it was paused.
Reset	This choice sends an abort message to the *Enterprise* executive, which terminates all the program's active processes. The Replay View is reset to its initial state. *Reset* and *Resume* are then replaced as menu choices by *Start*.
Step	Step can be used from the initial state to start a replay. The program will be launched and then stopped before the first Send Message event is processed. If used when a replay has been stopped (paused), the next event in the event queue will be processed. If this is a message-passing or Die event, the interface will send a replay message to the *Enterprise* executive to execute it.
Set Option	This is identical to the *Set Options* command in the Animation View. The only difference to point out is that the number of events to process in each animation loop in the Replay View is defaulted to 1, while the default in the Animation View is 5. For the breakpoint facility to exercise full control over replay, it is necessary to limit each animation loop to one replayable event.
Breakpoint Browser	This option will open a Breakpoint Browser, described below.

Aside from the main pop-up menu in the Replay View, there are specialized menus for assets and message queues.

6.3. Asset Menu: Sequential Debuggers

The pop-up menu that appears for an asset has a single option, Sequential Debugger, to attach a sequential debugger. This option appears only after a program has been started and then stopped. Moreover, it is necessary for the program to have been compiled with the debug option on. Sequential debuggers can only be attached to codable assets. Managers, which do not execute any user-defined code, cannot be debugged sequentially using the interface. The debugger is not directly associated with the user's original code, but rather with the code generated by the *Enterprise* pre-compiler containing modifications of the user's code. The intermingled *Enterprise* code and user-defined code is potentially confusing for the novice user. However, with a modest effort the user can effectively utilize the sequential debugger to locate difficult to find errors.

Currently, it is necessary for the user to execute the command *xhost* + *machineName* at the console to get remote debuggers to display. Or, if using Xauthority, have the *.Xauthority* file exported to remote file systems. Enterprise does export the DISPLAY environment variable from the local processor to all processes. However, the user is responsible for ensuring the DISPLAY value contains sufficient information to allow remote processors to resolve the location of the display. The user can specify a sequential debugger in the environment variable ENTERPRISE_DEBUGGER. If no debugger is specified, *gdb* is the default. To get the debugger window to pop up, it is necessary to Step the replay once. The relevant process halts when a debugger is attached. A breakpoint can then be set, say, on the function bearing the asset name. The debugger must be given the command to continue before the process will resume execution.

6.4. Event Breakpoints

Breakpoints can be set only at a high level, in terms of message-passing events. These breakpoints can be either unconditional for a particular event type at a particular asset grouping or conditional upon the values of any parameters which have been captured in the event file. When a set breakpoint is triggered, the guided replay stops just *before* the event is executed. It is then a simple operation for the programmer to single step through the event, examine the complete contents of a logged message, or attach a sequential debugger to any process for lower-level debugging.

A largely text-based Breakpoint Browser is coordinated with the graphical representation of the program while it is animating and replaying in the Replay View. This visualization technique is in accord with a core goal of *Enterprise*: to provide an intuitively comprehensible interface to an inherently complex and potentially confusing parallel architecture.

A breakpoint is defined for an individual codable asset, or for an arbitrary group of replicas of a codable asset. Composite assets, such as Departments, Lines, and Divisions, which have no code directly associated with them, cannot have breakpoints. Thus, a single breakpoint could be associated with two replicas of an asset, but not with two assets having different base names.

There are only four types of message-passing events relevant to breakpointing: a Send Message Event, a Receive Message Event, a Send Reply Event, and a Receive Reply Event. Every breakpoint is associated with one, and only one, of these event types, but not all event types are meaningful for a particular asset. For example, the first asset in a program cannot receive messages or send replies.

The assets that a given asset communicates with are its collaborators. For a Send Message or Receive Reply event type, an asset's collaborators lie below it in the asset diagram; for a Receive Message or Send Reply event type, the collaborators lie above the asset. A breakpoint must have one or more collaborators associated with it. As was the case with a breakpoint's assets, a breakpoint's collaborators must share the same base name.

Once a breakpoint's assets, event type, and collaborators are selected (as described below), a listing of logged parameter names and types is available from the system. A breakpoint's conditions can be any conjunction of conditional relationships between these parameters and user-supplied constant values. The conditional relationships supported by the system are '==' (equal), '!=' (not equal), '>' (greater than), '<' (less than), '>=' (greater than or equal), and '<=' (less than or equal). For instance, for a logged parameter `y` we could set the two conditions: `y >= 0` and `y < 10`. The breakpoint would then be encountered when both these conditions were satisfied for the specified assets, event type, and collaborators.

Finally, a breakpoint is not necessarily triggered each time it is encountered. The user can specify the number of encounters before a breakpoint is triggered. The default is to break on the first, and only on the first, encounter. If the user wishes to break on every encounter, an Always Break option can be checked. If the user wishes to break on the fifth encounter, the number of encounters can be set to 5. If the user desires to break on *every* fifth encounter, both the above settings can be made.

6.5. The Breakpoint Browser

The integrated tool for managing breakpoints is the Breakpoint Browser (Figure 6.1). Its overall design is similar to the Smalltalk System Browser. The top portion of the Browser consists of four hierarchically dependent list views for navigating through the breakpoint system. The bottom portion contains the mechanism for actually defining, editing, and accepting breakpoints. The Breakpoint Browser can be enlarged with proportional enlargement of the list views. It cannot be resized smaller.

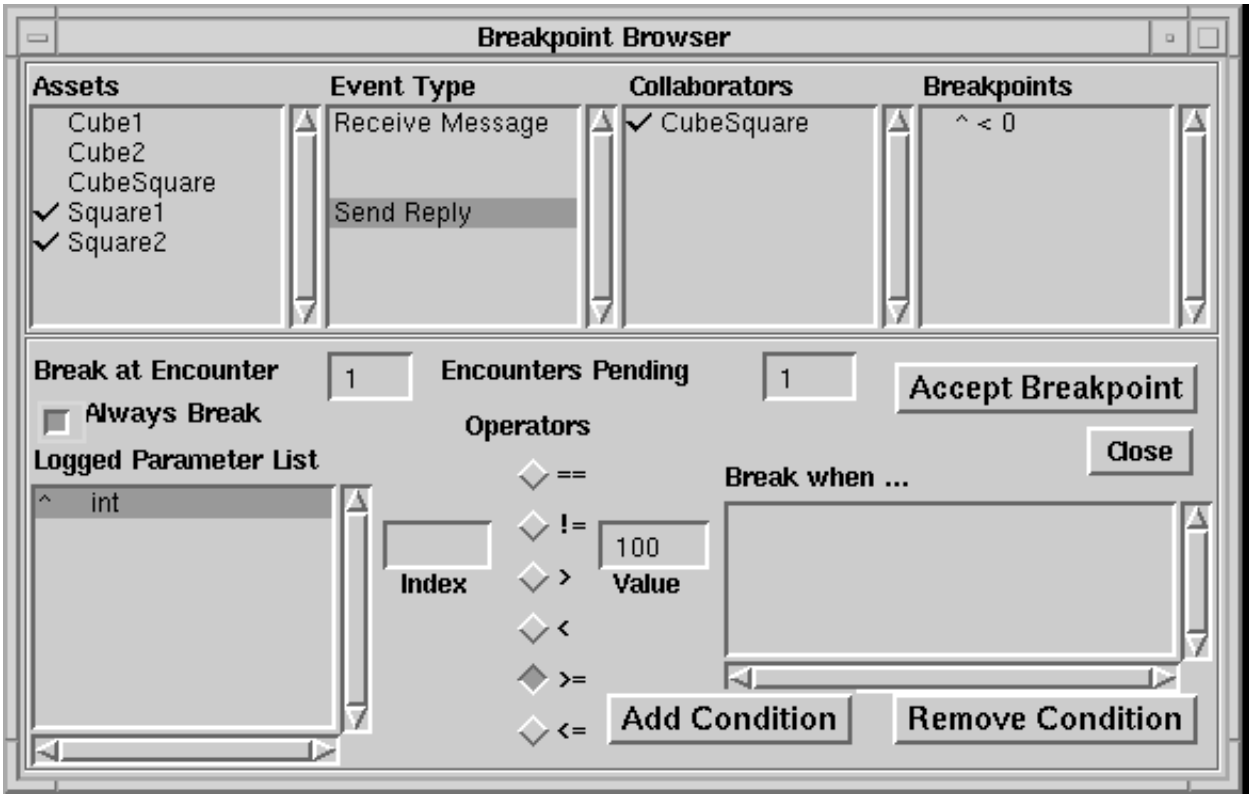


Figure 6.1 Breakpoint Browser

The first of the list views in the top portion of the Breakpoint Browser is the Assets list. This view always contains an alphabetical list of all the assets in the Replay View using their animation names. An asset's animation name is its base name, with an appended numerical suffix if the asset is a replica. These animation names are unique, regardless of replication nesting, and correspond to the names that appear in the Replay View. Multiple selections can be made from the list by clicking the <select> button when the cursor is over a list item. Selected assets are preceded by a check mark (√). Only assets with a common base name can be multiply selected. A warning appears if an incompatible multiple selection is attempted. Figure 6.1 shows a Breakpoint Browser with the assets *Square1* and *Square2* selected in the *Assets* list.

While the cursor is in the Assets list view, there is a pop-up menu for breakpoint deletion and asset selection. These menu options are context-sensitive, varying with the existence of breakpoints and the current selection of assets. The possible menu options are:

Remove all breakpoints
Remove all selected breakpoints
Deselect all
Select all replicas

Remove all breakpoints will remove all the set breakpoints for all the assets in the system. If there are no breakpoints in the system, this menu option will not appear.

Remove all selected breakpoints will remove all the set breakpoints for all the currently selected assets. If no assets are currently selected or if the selected assets have no breakpoints, this menu option will not appear.

Deselect all deselects all the currently selected assets. Alternatively, clicking on a selected asset with the select button individually deselects it. The Deselect all option does not appear if no asset is currently selected.

Select all replicas will select all the replicas of a currently selected asset. This option will appear only when an asset is currently selected and replicas of the selected asset are currently unselected.

The Event Type list, immediately to the right of the Assets list, is empty if no asset is selected in the Assets list. If an asset is selected, the Event Type list will display the types of message-passing events possible for the selected asset. In other words, an event type that yields no potential collaborators would not be displayed. For example, a Service asset can only have a Receive Message or Send Reply event type; the first asset in a program can only have a Send Message or Receive Reply event type. This event type determination is solely based upon the program's *Enterprise* design as reflected in the asset diagram, not upon the program code or the events actually logged in the event file.

The Event Type list is a list view that allows a single selection. If an unselected item is clicked on, it becomes selected (displayed in reverse video), and any previously selected item becomes deselected. Clicking on a selected item deselects it. The Breakpoint Browser in Figure 6.1 has Send Reply selected in the Event Type list.

The context-sensitive menu for this pane contains only two possible options:

Remove all breakpoints
Remove all selected breakpoints

The behavior of these operations is analogous to that of the Assets list menu options. Removal affects all breakpoints for the selected assets and the pertinent event type(s), irrespective of collaborators.

The Collaborators list, situated immediately to the right of the Event Type list, is empty until an asset and event type have been selected. When an event type has been selected in the Event Type list, all the potential collaborating assets are displayed in the Collaborators list. The rules for collaboration are as follows:

(1) For a Line, each asset, except the last, can send a message to the next asset in the Line. The last asset can send a message out of the Line if the Line is embedded in another Line with more assets remaining.

(2) For a Department, the Receptionist can send a message to all the other Department assets. All the assets in a Department, including the Receptionist, can send a message out of the Department if the Department is embedded in a Line with remaining assets.

(3) For a Division, the Receptionist can send a message to its Representative or Division component. All the assets in a Division can send messages out of the Division if the Division is embedded in a Line with remaining assets.

(4) All Receptionists, Individuals, and Representatives can send messages to all Services.

(5) If a message-receiving asset is a replicated asset (represented by a manager), the receivers of the message are taken to be all the replicas, rather than the replicated asset itself.

(6) If the receiving asset is a composite asset, the collaborator of a Send Message event is the Receptionist of the composite.

(7) If a receiving asset is an Individual, Representative, or Service, it becomes a collaborator of the Send Message event.

(8) The collaborators for a Received Reply event are the same as those for a Send Message event.

(9) The collaborators for a Send Reply event or a Receive Message event are the inverse of those of a Send Message event. That is, if `AssetB` is a Send Message collaborator for `AssetA`, then `AssetA` is a Send Reply collaborator and a Receive Message collaborator for `AssetB`.

At the time a message is sent to a replicated asset, it is not known which replica will ultimately receive the message. Although the ultimate destination could be determined in a replay situation by searching ahead through the event file, it is intuitive and logical to simply list the base name of the replicated asset, rather than all of the replicas individually. Selecting the replicated asset's name then effectively selects all the replicas as collaborators for the purpose of defining a breakpoint.

Like the Assets list, the Collaborators list is a multi-selection list view. All the selection operations and menu choices are identical to those in the Assets list. Breakpoint removal behavior, though not identical, is analogous. Figure 6.1 shows *CubeSquare* selected in the Collaborators list.

The Breakpoints list, the rightmost breakpoint navigation tool, is empty until a collaborator, or set of collaborators, is selected. Once the collaborator selection is made, an abbreviated list of breakpoints appears in the Breakpoints list if breakpoints exist for the chosen assets/event-type/collaborators selection. A breakpoint is identified and labeled by the first condition in its set of conditions. A breakpoint with no conditions is labeled as *Unconditional*. When a breakpoint is selected from this list, its full specification is displayed in the lower portion of the browser. This specification can then be edited as described below.

The selection mechanism and menu options are similar to those in the other multi-selection list views. Any number of breakpoints can be selected concurrently. However, if more than one is selected, the information in the lower portion of the browser reverts to the default settings, not a representation of any of the selected breakpoints. Multiple selection is only useful for quick selective removal of set breakpoints. Figure 6.1 illustrates a Breakpoint list with one unselected breakpoint.

There is a subtle feature regarding the removal of breakpoints involving replicas of assets or collaborators. If, for instance, a breakpoint is set for all replicas of an asset and all replicas of a collaborator, it will appear in the Breakpoints list for any sub-group of those replicas of assets and collaborators. If removal is then initiated through the Breakpoint list, the breakpoint will be removed only with regard to the selected assets and collaborators. It will still exist in the system for any unselected assets and collaborators left in its specification. If a breakpoint contains no assets or no collaborators in its specification, it is no longer in the system.

The lower portion of the Breakpoint Browser contains the breakpoint definition tools. These allow the user to specify the conditions for breaking and the number of times those conditions must be encountered before the breakpoint is actually triggered.

The field labeled Break at Encounter specifies the number of times this breakpoint's conditions must be satisfied before the breakpoint is triggered. The default setting is 1. The Encounters Pending field is read only. It is automatically set equal to the value of the Break at Encounter field when the breakpoint is accepted. During replay, when this breakpoint is encountered, the Encounters Pending field counts down by one. When Encounters Pending reaches zero, the breakpoint is triggered.

Once Encounters Pending reaches zero, the breakpoint is no longer active unless the user checked the box Always Break at the time the breakpoint was created. Checking Always Break automatically resets the Encounters Pending to its initial value after the breakpoint has been triggered and passed. Since the breakpoint is triggered and the replay stopped *before* the triggering event is replayed, the system must allow that breakpoint event to actually get replayed when replay is resumed. Therefore, to prevent deadlock, Encounters Pending is not re-initialized until after the event is executed.

As previously indicated, a breakpoint need not have any conditions attached to it. Such a breakpoint, labeled as *Unconditional* in the Breakpoints list, will be triggered on the n^{th} encounter of an event involving the selected assets, event type, and collaborators, where *n* is the value set for Break at Encounter.

A breakpoint can have any number of breakpoint conditions attached to it. Each condition is described by selecting a parameter from the Logged Parameter list, selecting an operator from the Operators radio buttons, entering an appropriate value in the Value field, and, if necessary, entering an integer index in the Index field.

The parameters logged in the event file for the event type in question are listed in the Logged Parameter list using (1) the parameter's name from the argument list of the called asset, (2) a description of the parameter's *logged elements*, if the parameter is an array, and (3) a description of the parameter's type. The return value of an asset call, which is unnamed, is identified by the caret symbol (^). The logged elements notation for arrays describes the indices of the array values which have been logged, as described above in Section 5.8. Following the C convention, the first element of an array is indexed by 0. Negative index values denote elements referenced from the end of the array. Thus, -1 refers to the last element; -2, the next to last. A range can be represented by a pair of indices in parentheses ( ). The logged elements notation allows any number of single indices and ranges to be included within square brackets [ ]. For example,

```
y[(0 2) 5 (-3 -1)]      array of int
```

denotes that the logged elements (integer values) of the array `y` are namely the first three elements, the *sixth* element, and the last three elements. The notation

```
y[(0 -2)]               array of char
```

could be used for logging a character string in C, omitting the terminating null byte, assuming that `y` is dynamically allocated to be the size of the string it contains.

To reference a value in an array, the user can enter an integer in the Index field. If an index is entered for a scalar, or if the index entered for an array is incompatible with the logged elements notation, attempting to add the condition to the list of conditions will generate an error warning. The invalid condition will not be added to the list. Since arrays can be dynamically allocated and of variable length, there is no way to check if an index is out of bounds when any of the *logged elements* or the index itself is referenced from the end of the array.

The relational operators (==, !=, >, <, >=, and <=) appear as radio buttons. Only one operator can be selected at a time.

To construct a valid condition, the programmer must make a valid entry in the Value field. If the type of the selected parameter is *int*, an integer (no decimal point) must be entered. For a *float*, the decimal point is optional. Scientific notation can also be used. For a *char*, the user can enter a printable character or the decimal ASCII code preceded by a backslash (\). If the logged parameter is a single range of *array of char*, a string of characters can be entered in the *Value* field. This string may contain non-printable characters written as a backslash followed by three digits representing decimal ASCII code. A backslash character in a string is represented by two backslashes. (See Section 8 for restrictions)

This mechanism of constructing a condition through list view selection, radio button selection, and input field validation makes it simple for the user to quickly create legitimate breakpoints. The possibility of introducing errors is minimized.

Once a condition has been constructed, it can be added to the breakpoint's conditions by clicking on the Add Condition button in the Breakpoint Browser. If the condition is completely and validly constructed, it will be added to the list view labeled Break when . . . . Otherwise, an appropriate warning message appears. Any number of conditions can be added in this manner. All the conditions in the list view are logically ANDed. If an OR relationship is required, separate breakpoints can be set up to achieve this relationship. This requirement is not restrictive: logically, any combination of ANDs and ORs can be reduced to disjunctive normal form, a disjunction of conjunctions of elementary conditions.

If a condition added to the Break when . . . list is not desired, it can be removed by selecting it and then clicking on the Remove Condition button in the browser.

After all the desired conditions have been added and the encounter setting made, the breakpoint can be accepted by clicking on the Accept Breakpoint button. When a breakpoint is accepted, three things occur: (1) the first condition of the breakpoint (or *Unconditional*) will appear unselected in the Breakpoints list; (2) the breakpoint settings in the Browser will revert to the defaults; and (3) an icon of a stop sign will appear beside the appropriate asset icons in the graphical replay view.

A breakpoint does not become an active part of the system until the Accept Breakpoint button is clicked.

When a single set breakpoint is selected in the Breakpoints list, the particulars of the breakpoint are displayed for editing. The encounter setting can be changed and conditions may be added or removed. When an edited breakpoint is accepted, the operation is equivalent to removing the previously set breakpoint and accepting the currently defined breakpoint.

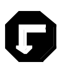
As pointed out above, when multiple breakpoints are selected from the Breakpoints list, the breakpoint definition portion of the browser reverts to default values. Breakpoint acceptance is not allowed in this circumstance. All breakpoints in the Breakpoints list should first be deselected before defining and accepting a new breakpoint.

A breakpoint can be deactivated, rather than removed, by setting the Break at Encounter field to 0, ensuring that the Always Break box is not checked, and then accepting the breakpoint. This results in retaining the breakpoint and all its conditions in the system, but the breakpoint will never be triggered. Such a deactivated breakpoint can easily be reactivated by again editing the Break at Encounter field and accepting the modified breakpoint.

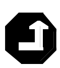
6.6. Graphical Display of Breakpoints

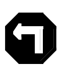
The management of breakpoints is simplified by the coordination among the display in the Replay View, the contents of the Breakpoint Browser, and the breakpoint state of the underlying model.

Any time a breakpoint is set, its existence in the system is portrayed as a red (black in monochrome displays) stop sign icon at a corner of the asset icon with which it is associated. The relative position of the breakpoint icon and a directional arrow in its center identify the type of event to which the breakpoint is responsive:

 Send Message (lower left corner of asset icon)

 Receive Message (upper left corner of asset icon)

 Send Reply (upper right corner of asset icon)

 Receive Reply (lower right corner of asset icon)

The presence of a stop sign icon indicates that one or more breakpoints of that particular event type are associated with the particular asset.

When an asset is selected in a Breakpoint Browser, its icon is highlighted with a wide red (black in monochrome) border in the Replay View. The zones for the breakpoint icons, which correspond to valid event types for a highlighted asset, are also outlined with a fine black line. Figure 6.2 shows *Square1* and *Square2* selected as assets.

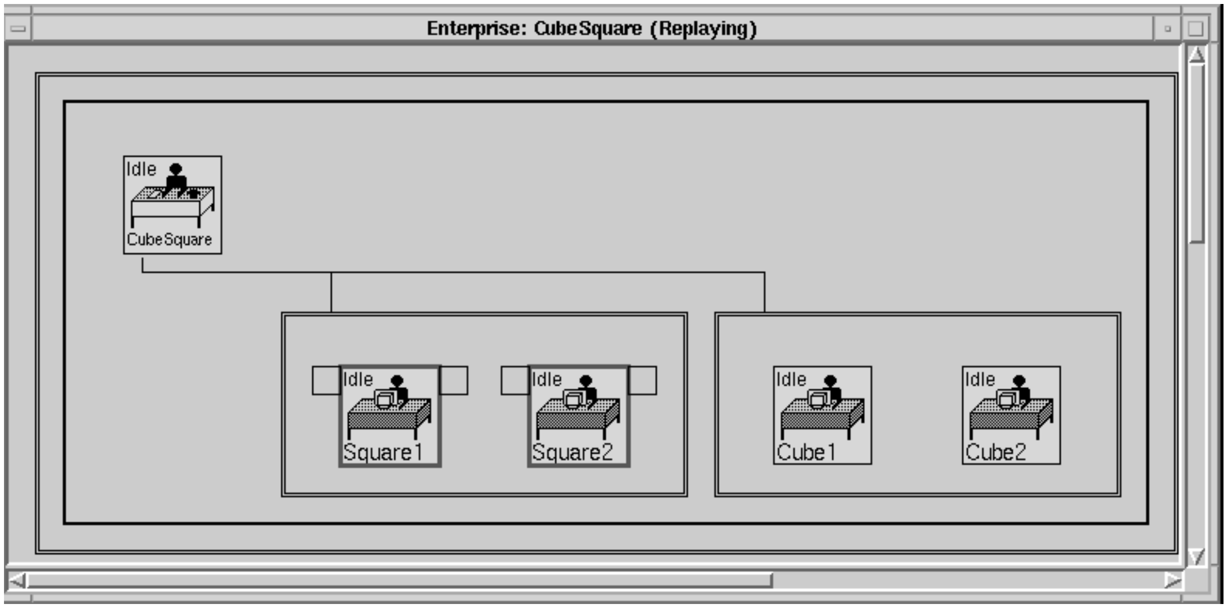


Figure 6.2 Replay View of *CubeSquare* with assets selected.

When a event type is selected, the breakpoint icon zone associated with that event type is highlighted with a wide black line. The unselected breakpoint icon zones lose their outlines. Figure 6.3 shows the replicas of `Square` with the Send Reply event type selected.

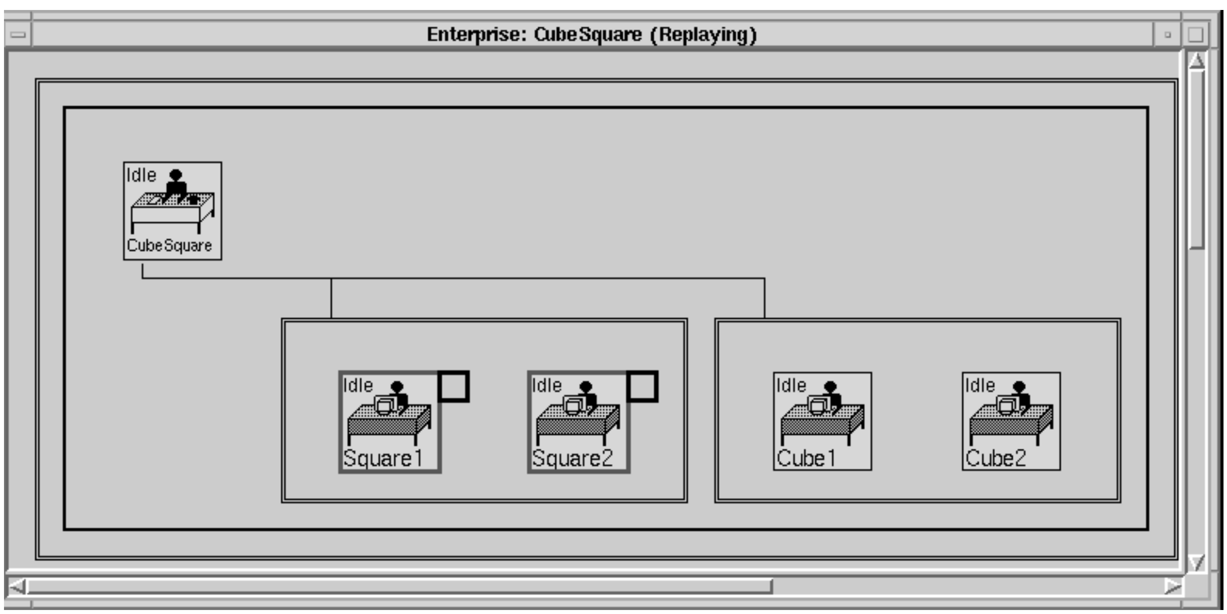


Figure 6.3 Replay View of *CubeSquare* with assets and event type selected.

When a collaborator is selected, a wide red (black in monochrome) line appears between the selected asset's breakpoint icon zone and the collaborator, suggesting the message path that would be followed from asset to collaborator, or vice versa. Multiple selections of assets or collaborators are displayed accordingly. Figure 6.4 illustrates this next step with *CubeSquare* selected as the collaborator for a breakpoint setup.

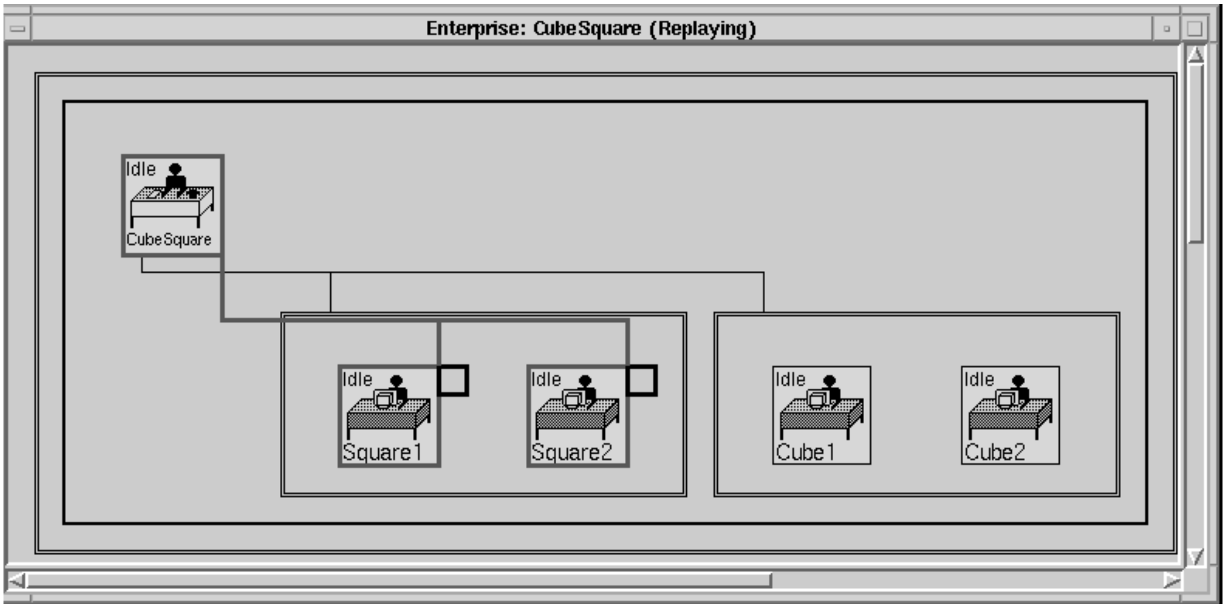


Figure 6.4 Replay View of CubeSquare with assets, event type, and collaborator selected

Figure 6.5 shows a program displaying the highlighting that corresponds with the selections in the Breakpoint Browser in Figure 6.1. To illustrate the different breakpoint icons, breakpoints have been set for various assets using all four types of message-passing events. `CubeSquare` has breakpoints set for Send Message and Receive Reply events. The `Square` replicas have breakpoints set on Send Reply events, and the `Cube` replicas on Receive Message events.

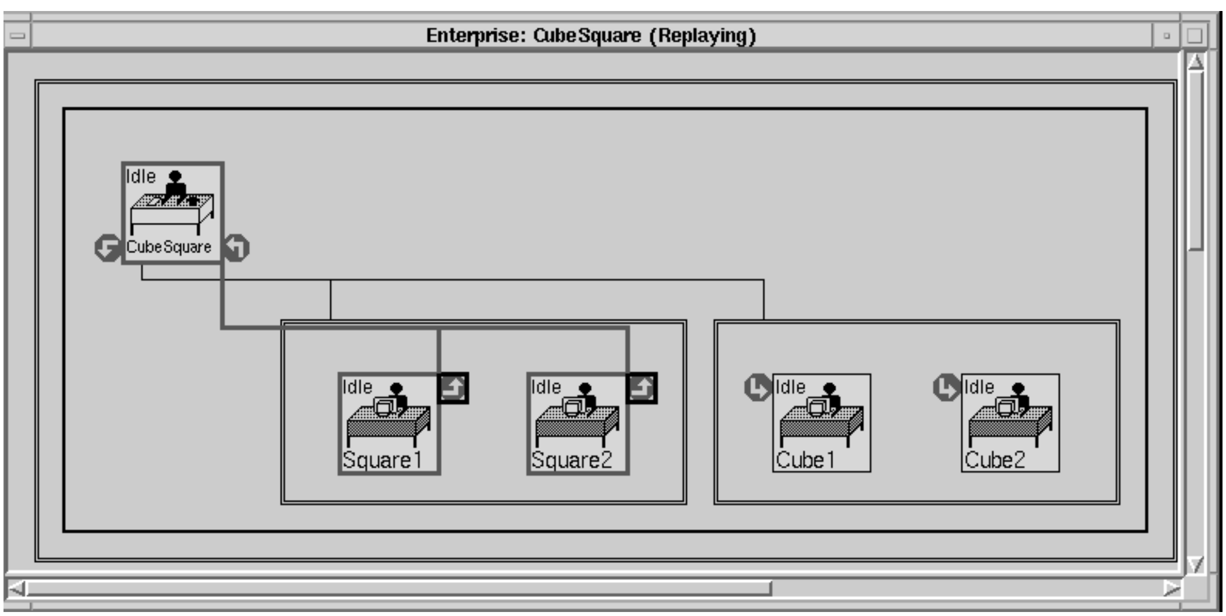


Figure 6.5 Replay View with breakpoints set.

Browser selections can be controlled by selecting icons in the Replay View, rather than items in the browser's navigation tools. An asset can be selected by clicking the *select* mouse button when the cursor is over the asset icon. Multiple asset selections are made by holding down the SHIFT key while making the selection.

The event type can then be selected by clicking in the appropriate outlined breakpoint icon zone. Finally, any number of collaborators can be selected by holding down the CTRL key while making the selection.

Any breakpoint that is set in the system can be selected by clicking on its associated breakpoint icon. If more than one breakpoint is associated with an icon, the user can repeatedly click on the icon to cycle through all the set breakpoints. As each breakpoint is selected, all its associated assets and message paths to collaborators are appropriately highlighted and selected in the browser's list views.

The system allows any number of Breakpoint Browsers to be opened simultaneously. But only one browser is considered active at any moment. The active browser is the last browser to be opened or the last to have had any selection made in its navigation list views or any of its action buttons pressed. Only the active browser has its state reflected in the display of the Replay View. Any time a selection is made in the replay view, it will be reflected in only the active browser. If an active browser is closed, the oldest remaining browser becomes the new active one. If no browser is active when a selection is made in the Replay View, a new browser will automatically be created.

6.7. Breakpoint Triggering

When a program is being replayed, each message-passing event in the event file is examined against the set breakpoints before the *Enterprise* executive is instructed to execute the event. Replay is initiated by selecting Start or Step from the Replay View menu. Step will process only one event from the event file. If this event is a Start Event, Die Event, or message-passing event, it will result in an instruction to the executive to proceed until another Die Event or message-passing event is imminent. If however an active breakpoint has all its conditions satisfied and its Encounters Pending is 1, the breakpoint will trigger and the event will not execute. Upon triggering, the responsible breakpoint will be highlighted in the Replay View, the active Breakpoint Browser will pop to the foreground (or be created if necessary), and the triggering breakpoint's specification will be displayed in it.

If the event type is Receive Message or Receive Reply, the specific message which triggered the breakpoint is still in the asset's input or reply queue. The user can open a Message Queue List on the appropriate message queue (select on the message queue icon) and then select the relevant message from the list view in the top pane of the Message Queue List. The message's logged parameters will then all be displayed in the text view located in the lower pane.

If the event type is Send Message or Send Reply, the user can single step through the triggering event, allowing the triggering message to enter its appropriate message queue. Once there, it can be examined in detail.

In the event that unlogged message parameters or local process state information is desired, the programmer may open a sequential debugger on an asset's process by selecting Sequential Debugger from the menu associated with the target asset.

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

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

(8) 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.

(9) It is not recommended that either *exit()* or *abort()* function calls be used in an Enterprise program. These functions conflict with the Enterprise program properly shutting down the other processes and communication system that compose the distributed parallel application. That is, it is non-deterministic whether or not the program will shutdown cleanly. One of the major irritations of parallel programs is cleaning up the multiple processes on multiple processors *by hand*. Using these functions raises the probability for experiencing such irritation. Later versions of Enterprise will include suitably modified replacements for these functions. Currently, the inclusion of either of these functions results in a compiler warning.

7.1 Enterprise directory structure

An Enterprise program is maintained as a directory of the program name and a series of subdirectories. Sometimes it is necessary to leave the interface to handle certain problems or make external processes. For example, the Mandelbrot and AlphaBeta demonstration applications have separate display programs. The user must build these applications in the Enterprise directory structure.

There are four important directories for the user to develop applications. They are *Assets*, *User*, *Include*, and *Data*. The *Assets* directory contains the source code for each asset in the Enterprise graph. An asset file has the file extent .e instead of .c. The *User* directory contains the user's source code that supports the asset code but is not parallelized directly. The *Include* directory contains the header files for both the asset and user source code files. The *Data* directory is used to store data files for the application.

Enterprise requires several directories to build the parallel application. They are *Graph, Src*, *Tmp*, *Obj*, and *Bin*. The Graph directory contains the various files used by the interface, including the textual representation of an Enterprise asset graph (the graph file, *.graph), the machine list used to spawn the processes (_ENT_MachineFile), logging format files (*.fmt), and event files (*.ev). The *Src* directory contains the modified asset code. This modified code is the parallelized asset code and will be eventually compiled to a specific machine architecture. The *Tmp* directory is used by Enterprise compiler for keeping the various scratch files. The *Obj* directory has subdirectories for each architecture where each compiled object file from the *Src* and *User* directories is stored. The *Bin* directory contains the executable code produced by the linker for each architecture. If a communication system requires the location of the executable in other than this directory, the executable is copied to the appropriate location[14].

At run-time the *Out* and *Err* directories contain the output of *stdout* and *stderr* for *each* process that Enterprise creates. Note, currently there is no *stdin* available for any Enterprise process.

8. Version 2.4 Restrictions

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

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

(2) When changing the replication of an asset from one to many, or from many to one, the asset must be recompiled. This is not a time-consuming task since the user's code is not affected.

(3) Currently, a homogeneous network of Sun 4, IBM, or SGI workstations must be used.

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

(5) String and char values in the Breakpoint Browser are not yet implemented. There is no error checking on entries for value and index.

(6) Futures are not allowed in the loop control statements *for*, *while*, and *do...while*. The reason is obvious; every time through the loop a (costly) check would be required to see if the future has returned. For example, if data is a future, then the following loop will not be allowed:

```
while( data[ i ] > 0 )
```

However, control statements such as, *if*, *switch*, or *return* do allow futures.

[14] The default location for PVM binaries is $(HOME)/pvm3/bin/$PVM_ARCH where PVM_ARCH is a PVM environment variable.

(7) Futures are not allowed to have side-effects. Our compiler can handle most side effects in futures correctly, but rather than possibly introduce an error in a program, we have decided to disallow all side effects. For example, if *data* is a future, then

```
i = data[ ++i ];
```

is not allowed.

(8) There is no stdin available for the distributed processes.

9. 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) New asset types are needed, 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.

10. 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.

11. 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 menu item will email us your message.