Design of Object-Oriented Distributed Simulation Classes

James D Schoeffler, Principal Investigator
Professor of Computer and Information Science
Cleveland State University
Cleveland, Ohio 44115

Final Report

Grant No. NAG 3-1441

October, 1992 to November, 1995

Design of Object-Oriented Distributed Simulation Classes

James D Schoeffler, Principal Investigator
Professor of Computer and Information Science
Cleveland State University
Cleveland, Ohio

Final Report
Grant No. NAG 3-1441
October, 1992 to November, 1995

Summary

Distributed simulation of aircraft engines as part of a computer aided design package being developed by NASA Lewis Research Center for the aircraft industry. The project is called NPSS, an acronym for "Numerical Propulsion Simulation System". NPSS is a flexible object-oriented simulation of aircraft engines requiring high computing speed. It is desirable to run the simulation on a distributed computer system with multiple processors executing portions of the simulation in parallel. The purpose of this research was to investigate object-oriented structures such that individual objects could be distributed. The set of classes used in the simulation must be designed to facilitate parallel computation. Since the portions of the simulation carried out in parallel are not independent of one another, there is the need for communication among the parallel executing processors which in turn implies need for their synchronization. Communication and synchronization can lead to decreased throughput as parallel processors wait for data or synchronization signals from other processors.

As a result of this research, the following have been accomplished. The design and implementation of a set of simulation classes which result in a distributed simulation control program have been completed. The design is based upon MIT "Actor" model of a concurrent object and uses "connectors" to structure dynamic connections between simulation components. Connectors may be dynamically created according to the distribution of objects among machines at execution time without any programming changes. Measurements of the basic performance have been carried out with the result that communication overhead of the distributed design is swamped by the computation time of modules unless modules have very short execution times per iteration or time step. An analytical performance model based upon queuing network theory has been designed and implemented. Its application to realistic configurations has not been carried out.

1. Introduction

The Numerical Propulsion Simulation System (NPSS) project has as its objective the construction of a generalized analytical framework for studying the overall behavior of an

aircraft engine under both transient and steady-state conditions. Simulation of engines and their environment is a computationally complex and expensive component of the framework. NPSS envisions a distributed simulation of an engine in terms of a set of components implemented as objects in a language such as C++.

The objects are not independent of one another of course because they share interfaces. The pressures and temperatures at the outlet of one component must be identical to the corresponding variables at the inlet of a physically connected component for example.

The distribution of components implies that components execute (usually) in different address spaces or machines. In the simplest case, the simulation advances one iteration at a time (no time dependence) or one time increment at a time. In either case, it is necessary for components to share data which is coordinated by time or iteration.

Coordination and distribution imply communication in the form of messages from one component to another with the coordination being enforced by the availability or lack of availability of the data from the other component. Hence inter-module communication of data is central to the distributed simulation. Many other fields have design, analysis and simulation problems with the same set of needs.

There are three essential characteristics required of the objects to meet these needs.

1. Their structure must be such that they may be written independently of one another so that different components may be substituted, simulated, and evaluated. Furthermore, the structure must be such that the creator of the object can focus on the physics of the object and not the computer science and implementation considerations. Furthermore, each object may represent its engine component at varying level of fidelity (e.g., one dimensional versus three dimensional model). Hence the creator of an object must not assume knowledge of the details within other objects. The interface between objects must be such that differences in fidelity can be handled. For example, two adjacent components may use two different numerical grids to discretize variables such as pressure or temperature. The transformation of these grids must be possible at the interface. The interface between a component modeled as a one dimensional object and another modeled as a three dimensional object must transform the data for data-sharing purposes. This general requirement is termed "zooming".
2. It must be possible to distribute the objects representing the components to be simulated among separate but communicating processors to take advantage of parallel computation without change to the objects and without customization of software. Hence inter-object communication must not require special code dependent upon the distribution.
3. Objects must execute as thought they were truly running concurrently with one another without the need for an overall control program which understands the objects, their interconnection, and their distribution in order to properly sequence them.

The purpose of this research was to present a framework for objects which meet the essential characteristics listed above. Section 2 of this report discusses the decomposition of the overall system into concurrent communicating objects, the required internal structure of these objects, and the resulting computational model for an object. Section 3 discusses how the introduction of "connector objects" can provide the basis for distribution of the objects in a general way. Section 4 describes an execution model which is used in each process to provide the equivalent of concurrent execution. Section 5 describes an implementation of this approach and its results.

2. Decomposition into concurrent communicating objects

An object representing a physical component to be simulated is modeled conceptually as shown in figure 2.1. The module is shown with distinct inputs and outputs, each of which represent ports or connection points so that objects can be interconnected. The figure uses the term "module" to differentiate it from other objects (such as connector objects). By an input or output port is meant a path along which **data objects** which represent a group of data items which correspond to the variables at an interface between this component and another connected to it. Sharing of data is then taken to be sharing of data objects all of whose components represent variables calculated at a given time instant or iteration.

Similarly, outputs represent ports through which this module object can send data objects it has calculated at a given time instant or iteration. It is important to understand that the module itself must not know about the source of data objects it receives or the destinations of data objects it creates. It is the responsibility of the simulation user to interconnect modules for a given simulation. The form of this interconnection is described in the next section.

Objects represent engine modules and calculate pressure, temperatures, flows given input conditions to the module. The module is modeled in single or multiple dimensions using either steady-state or time-evolving relationships. The model interconnects objects two ways:

1. objects whose inputs are actually outputs of another object are interconnected in the sense that an update of the former object at a given time step or iteration cannot take place before the latter object output has been obtained.
2. solver objects are used to break closed loops which occur in (1) by supplying inputs to one module object in the loop based upon previous values of the output of the other module which must be the same as the supplied input when iteration converges.

As a basis for design, the MIT "actor" model has been developed for the purpose of easy distribution of the modules. Actor-object characteristics are:

1. Each actor acts as though it has its own thread of control

2. Each actor acts as though it has its own queue of messages which it processes one at a time (possibly in priority order) in a run-to-completion manner in the sense that a given actor-object does not start processing a second message until the processing of the first message has been completed. Note that this does not preclude the task in which the actor resides being blocked in favor of other tasks.
3. Processing is depending upon the state of the actor-object and may include change of state.

These characteristics greatly simplify the scheduling of module executions whether they are on the same machine or different machines-a key requirement for this simulation. The actor model associates a single thread of execution with each object. As a result, each module (actor) is assigned to a UNIX process in a given network of machines. There may be multiple processes per machine and multiple actors per process with the division arbitrary.

A control program which spawns the processes in the various machines and controls and sequences the actor objects through control of their messages containing outputs calculated by each module has been created. The C++ program has been extensively tested and modified to fit the objects of the distributed simulation. Its particular strengths lie in its control of the actors, handling of messages; its use of "connectors" to dynamically interconnect the modules into an arbitrary configuration; its ability to arbitrarily allocate the distribution of modules in response to user request; and its ability to automatically start up the parallel execution and coordinate through its completion.

The behavior of the module for one iteration is taken to be:

1. Request input data objects.
2. Wait (block) until the data objects have arrived.
3. Compute the data objects which represent outputs of the computation.
4. Send output data objects in response to requests.
5. Advance time/iteration and repeat the sequence.

The "public" structure of the module object which realizes this behavior is shown in figure 2.2. There are shown the object methods which support the above behavior. The initialization method resets the internal data members of the module including time and iteration number; the execute method carries out the main computation of the module and produces the output objects; the message processing method responds to messages of know type such as requests for module state or module statistics; the buffer to input data method transfers data from a received data object into internal data of the module; and the output data to buffer method transfers internally calculated data into data objects for output to other modules.

The interface is said to be "public" because these are the only methods that a module creator must write. The module creator decides what internal data is needed, and how the

computation is to be carried out, the essential part of the module, and implements this in the execute method. The module creator must also write the service methods which support the key execute method.

A major characteristics of the module structure is the communication via data objects. Such objects, because they are objects, can handle problems such as data transformations because of machine-dependent data storage problems and problems of error detection (wrong data object transmitted/received). Because data is shared by adjacent modules, it is necessary for the form of the data objects which contain the shared data to be defined to the system. Hence interconnection carries the implication of modules and ports which communicate and also the contents of the data objects which are communicated. Obviously it is not possible to modularize a simulation if one object does not produce data necessary for the other modules. Hence the identification of the shared data is not a problem. The form of the shared data however is an important consideration. The essential characteristic called "zooming" summarizes the control over this aspect: modules (through their buffer input and buffer output routines) or interface objects (via attached transformation or zooming methods) can (and sometimes must) transform the data to the internal form desired.

In addition to the public methods, there is a major internal private method which enforces the proper sequence of the calculation and calls the above methods. Figure 2.3 shows "state" of the module that is used and the enforced sequence. Note that the module then proceeds through a series of computations interspersed with waits. Hence each module executes as though it were a separate process. Section 4 discusses how this is actually carried out.

The major point of this section is the structure of the module which allows the creator of the module to divorce the essential design of the module's computation from the details of interconnection, execution, sequencing, and communication.

Not discussed here is the problem of interconnected modules which form tight closed loops. This requires the introduction of modules, called "solver modules" whose methods are identical but whose state sequence is slightly different. They pose, however, no different work for their creation.

3. Connectors as the basis for efficient inter-object communication

A major objective is to divorce a module from its interconnections, especially the possible need for multiple destinations of its outputs, source of inputs, and control over requesting and synchronization of delivery. To this end, all of this is done by standard internal private methods of the module object and do not concern the designer who is responsible only for creating the output data objects and using the input data objects.

The module private functions make input requests through each input port and respond to requests arriving at output ports. Connector objects are attached to each port for the

purpose of handling the configuration of the modules at run time (at run-time, the interconnection must be established), and the delivery of messages (including information which indicates that data is not yet ready and hence the requester will have to wait). Furthermore, the connector objects must do this efficiently. If two communicating modules are in separate machines, the data object must be communicated by messages. If two communicating modules are in the same process in a single machine, the data object must be communicated by simple function call. Thus the efficiency must be the same as for a communication/interconnection strategy which is planned in advance except that it can't be known in advance.

Figure 3.1 shows the connector objects which realize this interconnection. A **source** connector is attached to output port and a **destination** connector to each input port of each module. Since outputs may be connected to an arbitrary number of input ports, the source connector is actually a collection of objects. The net result is that each communication path starts at a source connector and terminates at a destination connector. Methods of the connector provide requests for data and calls to receive a data object. These methods are not called by the module creator in defining the module methods discussed in the previous section, but instead are used by the internal private methods of the module.

Notice in figure 3.1 that a separate interprocess-communication (IPC) object is attached to each connector. Upon startup, the objects are instantiated in each process in each machine. Configuration information is transmitted at run-time from the user (through a graphic user-interface or a batch file depending upon the user's choice) so that the choice of module objects to be instantiated is known and their distribution among machines. The location information is used by the connector to instantiate either a local-IPC object or a remote-IPC object each of which is a C++ class derived from an abstract IPC class. The methods of the IPC derived classes are virtual so that polymorphism permits the modules and connectors to use their methods with no knowledge about whether the connection is local or remote.

Figure 3.2 shows the effect of a communication between two modules in the same process (as would be the case in an integrated implementation). All transfers are by function call. Hence the requester module receives an ACK or NAK back from the connector depending upon whether or not the requested data object has been delivered. If the data object was ready for delivery (the source object was then waiting for all requests to be received before it starts another iteration), the request causes the data object to be transferred to the requesting module via a call to the input methods of the module and then returns an ACK to the requester. If the data is not available (the modules are asynchronous), a NAK is returned and the requesting module blocks until the object is received. Once requested, the module need not request it again as the request is noted in the source connector. Hence when the source module does complete its calculation and produces the output data object, it is immediately transferred to those modules which have already requested it. The arrival of that data object then changes the state of the requesting module so it can unblock and proceed (see section 4).

Figure 3.3 shows communication between two remote modules. In this case, the IPC objects automatically return a NAK to the requester because both the request and the data object return must be done through message passing. Such communication takes on the order of a millisecond or so and it is important to keep processors as busy as possible. Blocking the requesting module permits its machine to be allocated to another module in the same machine (if any). Meanwhile, the IPC formulates a message and transmits it. Its arrival is handled similarly to the direct transfer case, with the request being notes if the data is not available, or the data being transferred if it is. In the latter case, the module uses its output method to send the data to the connector and unaware that the IPC of the connector will then pack the object into a message for communication to the remote machine. The arrival of the message to the source module is made transparent by having the IPC of that connector construct the data object from the message and then pass the data module to the input method of the module just as though it had been directly passed.

Because it is desirable to allow any distribution appropriate to a given simulation, it is desirable to permit one, many, or even all modules and objects to be in one process. The latter case corresponds to an corresponding integrated simulation program. Because modules are concurrently and because inter-process message passing in workstation environments is process-to-process, it is necessary to have an execution control model to handle messages and multiple concurrent modules. This is discussed in the next section.

4. The execution model for the simulation

Distribution of the modules, connectors, IPC objects, and other objects is based upon the desired distribution of the modules themselves. Connector and IPC objects are always associated with a module. Hence the distribution results in a set of **processes**, with processes assigned to specific machines. A process is the basic scheduling unit in a computer. Within a process, however, there may be one or more modules. Since modules are explicitly designed to execute concurrently, it is necessary that there be execution control within a process. Without this control, a module which blocks waiting for input data for example, would block the entire process and hence the concurrent modules, thereby destroying the utilization of the parallel computers.

Our model of the concurrent modules is essentially the MIT Actor model which takes the object to execute concurrently with other actors (objects) each having its own "thread" of execution. This allows the module to block its own thread without blocking the threads of concurrent modules.

Simulation module objects described earlier are specifically designed so that they can be easily and efficiently managed by a control section of a process so that the set of modules. In particular, all I/O operations are carried out through calls to methods which do not block the module but rather return to the control section of the process. Actually the group of modules, connectors, and other objects are all contained within another objects,

the **module group** object, which manages the collection and contains methods for controlling the concurrent execution of modules within itself as though they were executing in separate threads.

The module group maintains a **ready list**, a list of modules waiting to compute but not containing those blocked waiting for input data objects or waiting for output data object requests. The module group execution method simply selects a module from the ready list, and calls its execution method. that module runs one phase of its activities and then returns to the module group execution method indicating whether it can be returned to the ready list or not.

Messages are passed from process to process by the underlying operating system and communication software. Hence messages arriving for a module are actually arriving to the module group object. Prior to scanning the ready list and allocating the computer to a module, any waiting message is examined to determine the destination module, and then transferred to that module by calling a message handling method of the input connector of that module to which the message is addressed. The connector method then calls module methods to transfer the data object into the module **without actually making the module active** (i.e., scheduling it for execution). It does this by calling internal methods of the module and connectors designed to be **passive methods**, that is, designed to be called whenever the active methods (the computational methods called when the module thread is scheduled).

Because execution of all concurrent modules is controlled by the module group and because modules return to the module group at controlled times, the use of passive methods is safe and every efficient. No special concurrency control mechanisms have to be invoked.

5. Experience and conclusions

The above design has been implemented in C++ and evaluated on a local area network of RISC processors. The overall object diagram for the system is shown in figure 5.1. Application modules are linked into the program along with the standard modules. Copies of the program are started on all machines cooperating in the simulation and then appropriate application modules dynamically instantiated and interconnected at run time. Tests of given sets of modules distributed in a variety of possible configurations from fully integrated to fully distributed demonstrate the practicality of the design. A detailed performance model has been constructed for the purpose of predicting the computing time of a set of modules given their distribution. This model incorporates the detailed state of each solver and module and of necessity includes overall network states. Because of its size, the model is limited to about 5 or 6 components. I intended to use the model in choosing the distribution of modules among work-stations once a user selects as set of modules for simulation. Since completion of that work, however, I have come to believe that a performance model based upon Mean-Value-Analysis and which includes queuing centers modeling both hardware modules (work stations) and software modules

(simulation components) would be feasible (removing limitations on number of modules) and more desirable (because of lesser computational time). Although the research contract has expired, I am pursuing this model and will make results available to the NPSS project.

5. Unsolved Problems

The results of this research indicated that an actor-based model can provide a flexible structure on which a distributed simulation can be based. In particular, it demonstrated that:

1. A simple specification of how an arbitrary module may be implemented and added to the system may be provided. In effect, a user written module need provide the methods of Figure 2.2

2. The specification of location of modules for execution can be specified arbitrarily at run time. In the test implementation, all classes were linked into a single executable program which is initiated on all machines on which modules are to run. The program then reads a simple shared file indicating the module name and class and machine on which it is to run. The program then instantiates the objects corresponding to these classes and they interconnect automatically at run time. This implies that the design is quite compatible with a Graphic User Interface that allows an engineer to specify an engine configuration (including distribution on machines) followed by immediate execution. Furthermore, an additional step that could optimally distribute the selected modules for load-sharing purposes would be easy to add.

3. The test implementation demonstrated that the efficiency of the interconnection is adequate, especially when individual modules have any significant amount of computation per iteration. This is expected to be the case especially for two and three dimensional engines models and transient analysis.

However three problems have not been addressed and need further research. They are:

1. The test implementation requires all modules that are to be included in the simulation to be pre-linked into a single executable program that can run on all machines in the distributed computation network. This is not a fundamental limitation. All that is required is that each machine in the network be able to run an executable program which contains the overall control logic and those classes which correspond to modules which will actually execute on the network. Nonetheless, this means that the location of the modules which are "special" in some sense must be known and a program linking step taken before execution. This would be quite satisfactory for certain situations such as a proprietary engine component. In fact, this would provide protection to a user for such components. Since the only engineers who add modules would have to be accomplished programmers, this step would be feasible.

It would be desirable to study how easily specialized versions of the programs could be created without interfering with the interactive use of the simulation tool.

2. The test implementation assumed that both module and solver classes are created and pre-linked into the executable program. However, it seems desirable to allow the engineer setting up a simulation to interact with it at the level of specifying special rules by which individual variables are updated by the simulation and the solver. This in effect means that the engineer must be able to add relatively simple statements to the program to accomplish this.

   It would not be difficult to provide "hooks" which allow this by adding functions which are called at key points in the execution cycle. Such functions would normally be empty and simply do nothing but could easily be overridden by the engineer in order to add additional statements to the program.

   Unfortunately it appears that this would have to be done by one or the other of the following methods:

   1. Allow the addition of statements but then require a re-compilation and linking of the executable program. This is not desirable because of the problem of requiring the engineer to be an accomplished programmer even though the statements being added are relatively simple. This is because of the necessity of understanding program structure (what and where the hooks are) and the problems of referencing variables within in the existing program. It is easy to imagine programming errors produced by the compiler and linker which would be baffling to a non-programmer engineer.

   2. Provide an interpreter which allowed the engineer to enter statements as text and which are then carefully interpreted by the interpreter at execution time. This interpreter would have to be carefully written so that understandable error messages were provided to the using engineer.

      It would be desirable to investigate the efficiency and adequacy of these and perhaps other methods of making the simulation flexible in the sense of customizing solvers without requiring complete coding of new solver modules.

   3. The automatic sequencing of the modules which guarantees that no module will execute until its inputs have been obtained is a key result of this research. However this also implies that there are no implicit loops in the specified simulation. Since the structure of the simulation is being set by an engineer interactively, this condition may not hold.

It is easy to determine that an implicit loop exists and warn the engineer. The solution is usually to add a solver which breaks the implicit loops. However it is often common practice to simply ignore the problem and use data that is one time-period old because after an iterative simulation converges, the one-period-old data and current data are the same anyway.

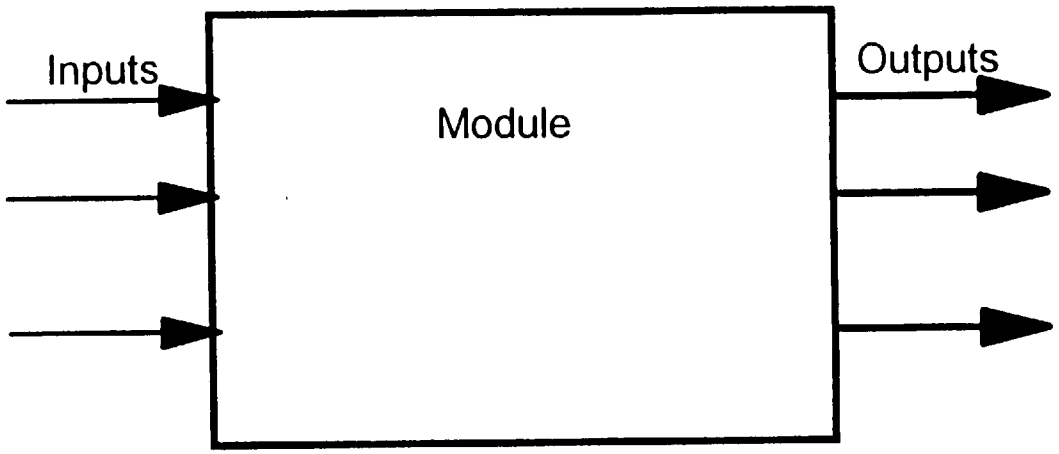
The test implementation does not allow this. It would be desirable to investigate how such situations could best be handled in a general way. Alternatives identified are:

1. Allow the engineer to specify the sequence of updating of modules arbitrarily. The simulation would then warn the engineer of the implicit loop but agree to run the simulation if the engineer indicated that the problem was to be solved by ignoring the loop and following the given update sequence. The simulation would have to be aware of this change and override the state logic. It is not clear whether or not this is a desirable solution.

2. The simulation could add an arbitrary solver to break the loops after warning the engineer of the implicit loop. The exact structure of such solvers and their effect on efficiency needs further investigation.

Figure 2.1

The Module Class



Messages (in and out) contain
Data objects

Figure 2.2

A Component as a C++ Class

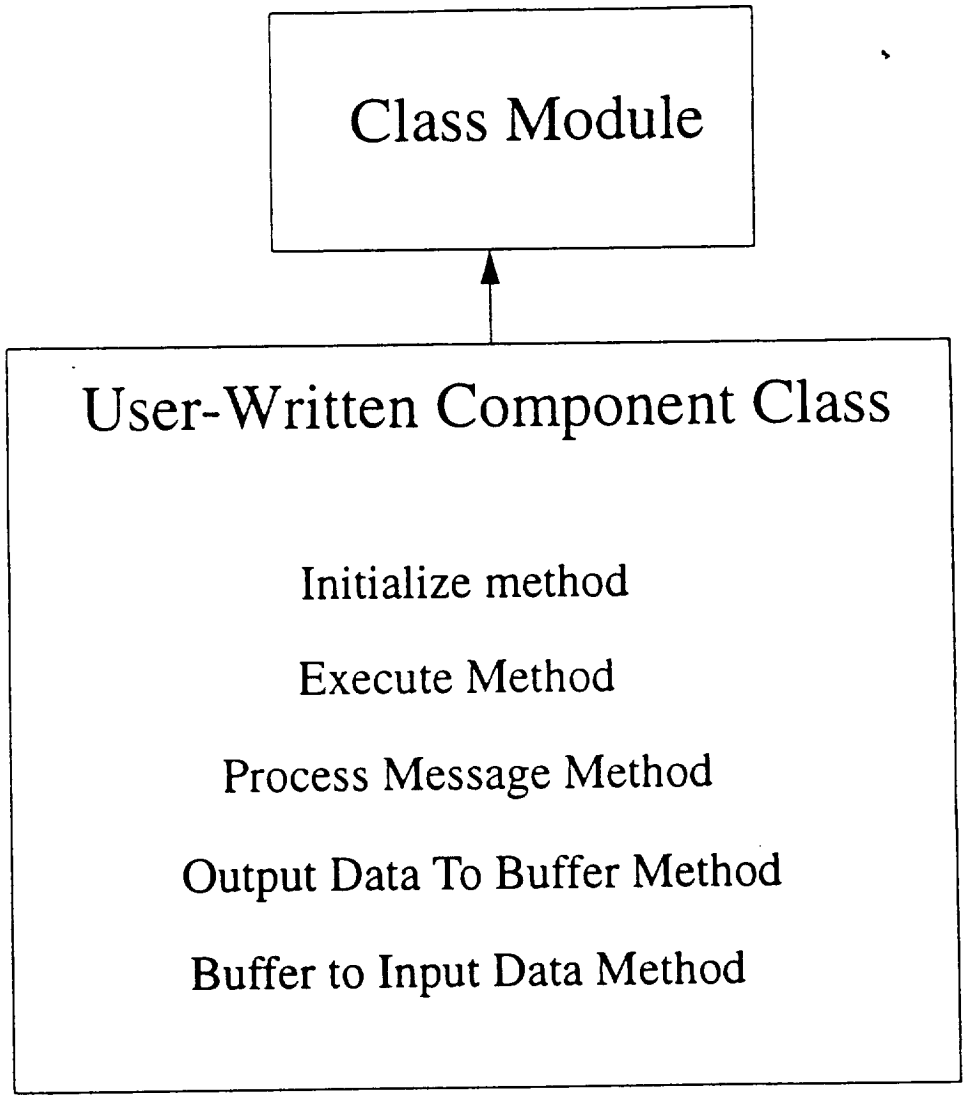


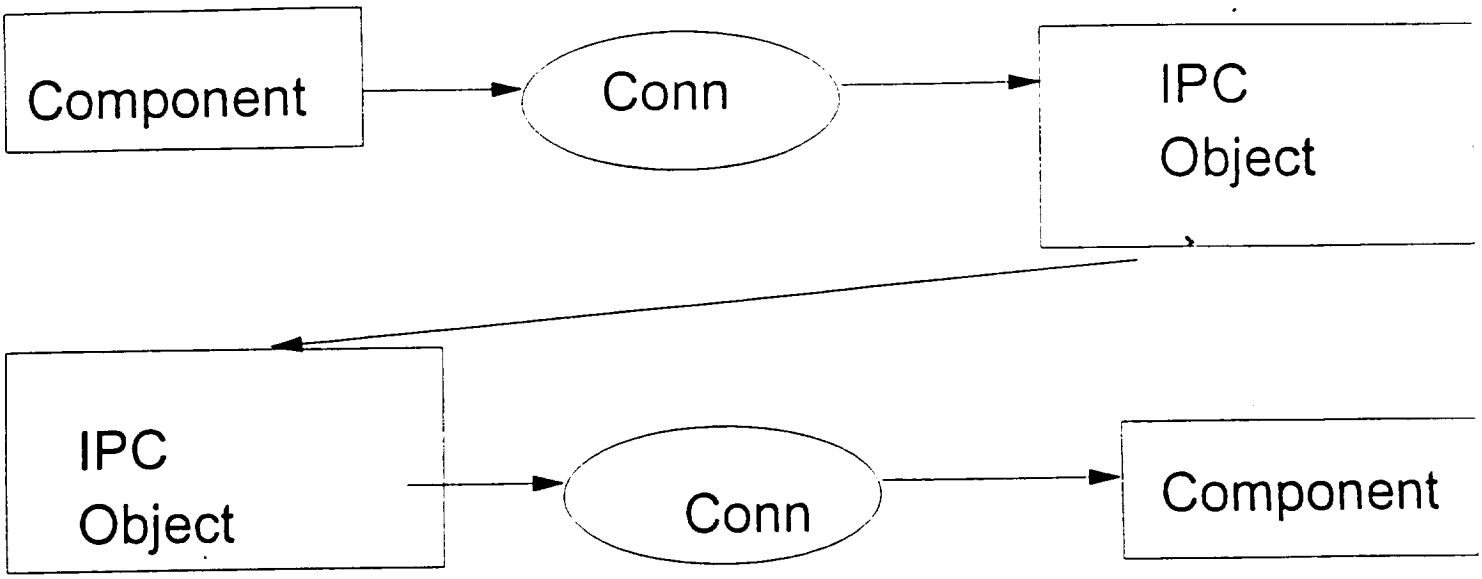
Figure 2.3

The State Sequence of a Component Module

Dispatch State	Sub-State	Action
(time/iteration is K)		
READY	Ready-No-Input	
		Request input Data objects
BLOCKED	Ready-No-Input	
		Input data objects arrive
READY	Ready-Input	
		Compute outputs Attempt output of data objects
BLOCKED	Block-Output	
		Send output data objects as requested
READY	Block-Output	
		Increment time and/or iteration to K+1
READY (time/iteration is K+1)	Ready-No-Input	
		.....

Figure 3.1

Connector Concept



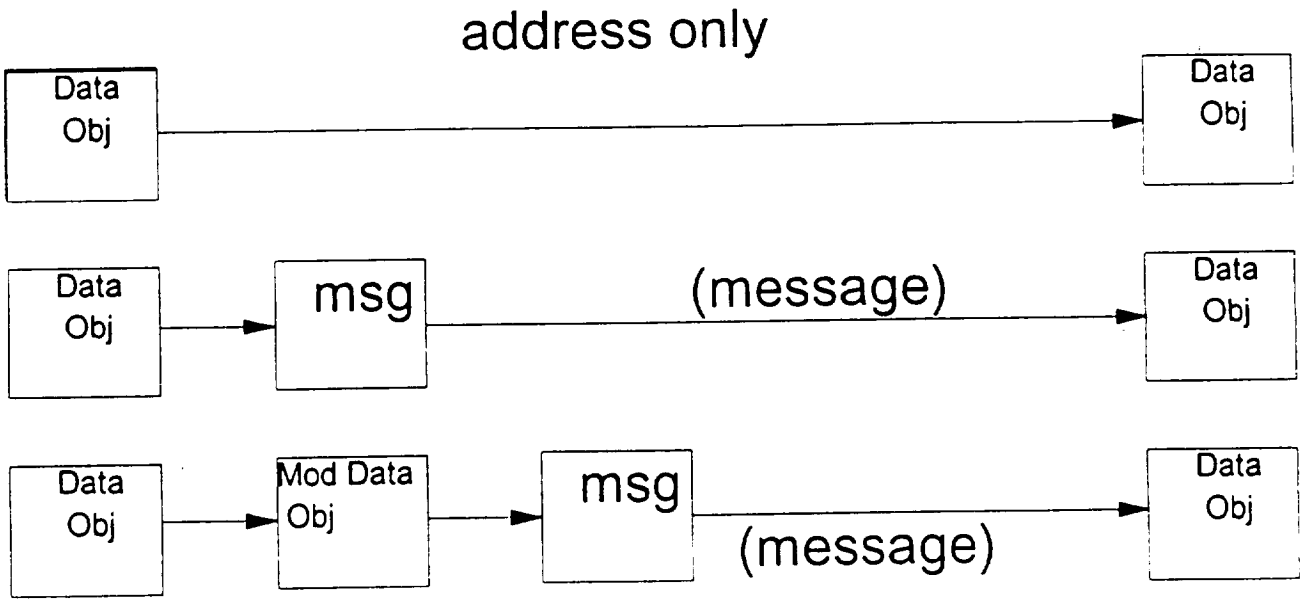


Figure 3.2

Inter-Module Communication
Both modules in same process

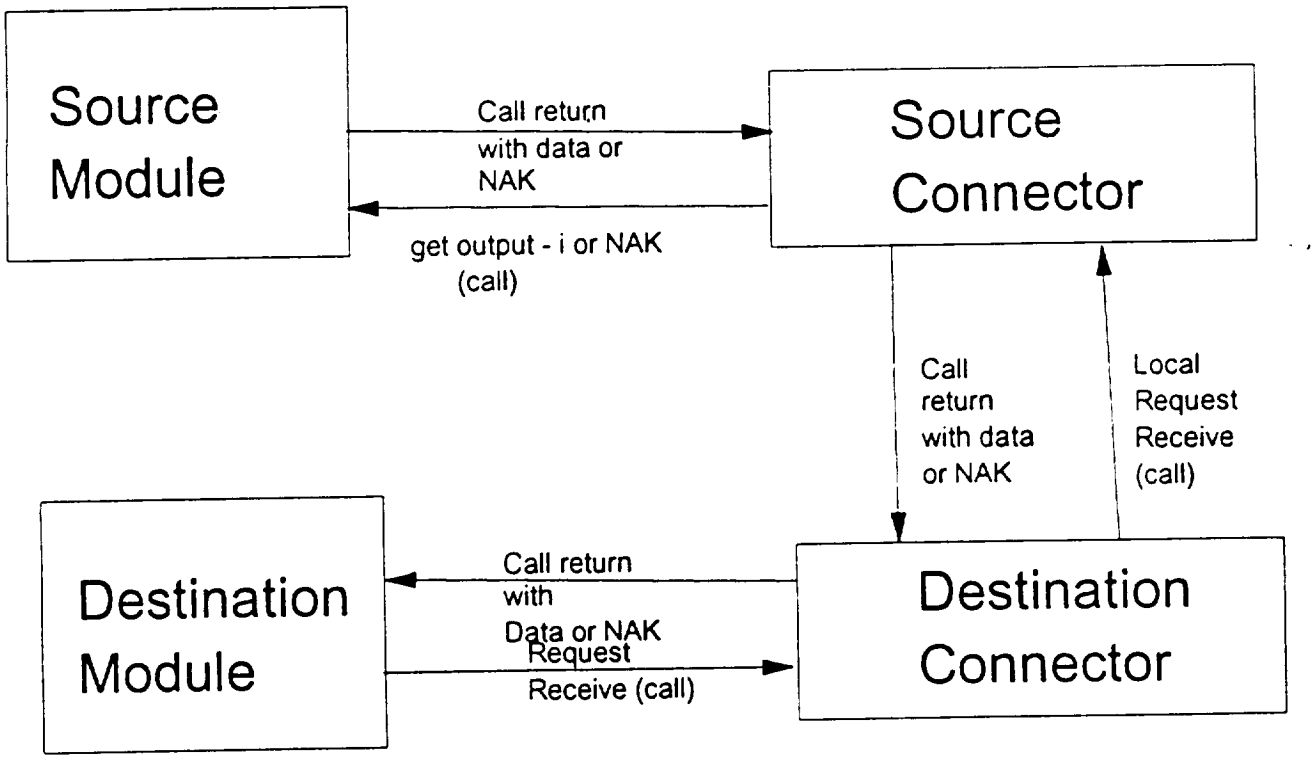


Figure 3.3

Inter-Module Communication
Modules in separate processes

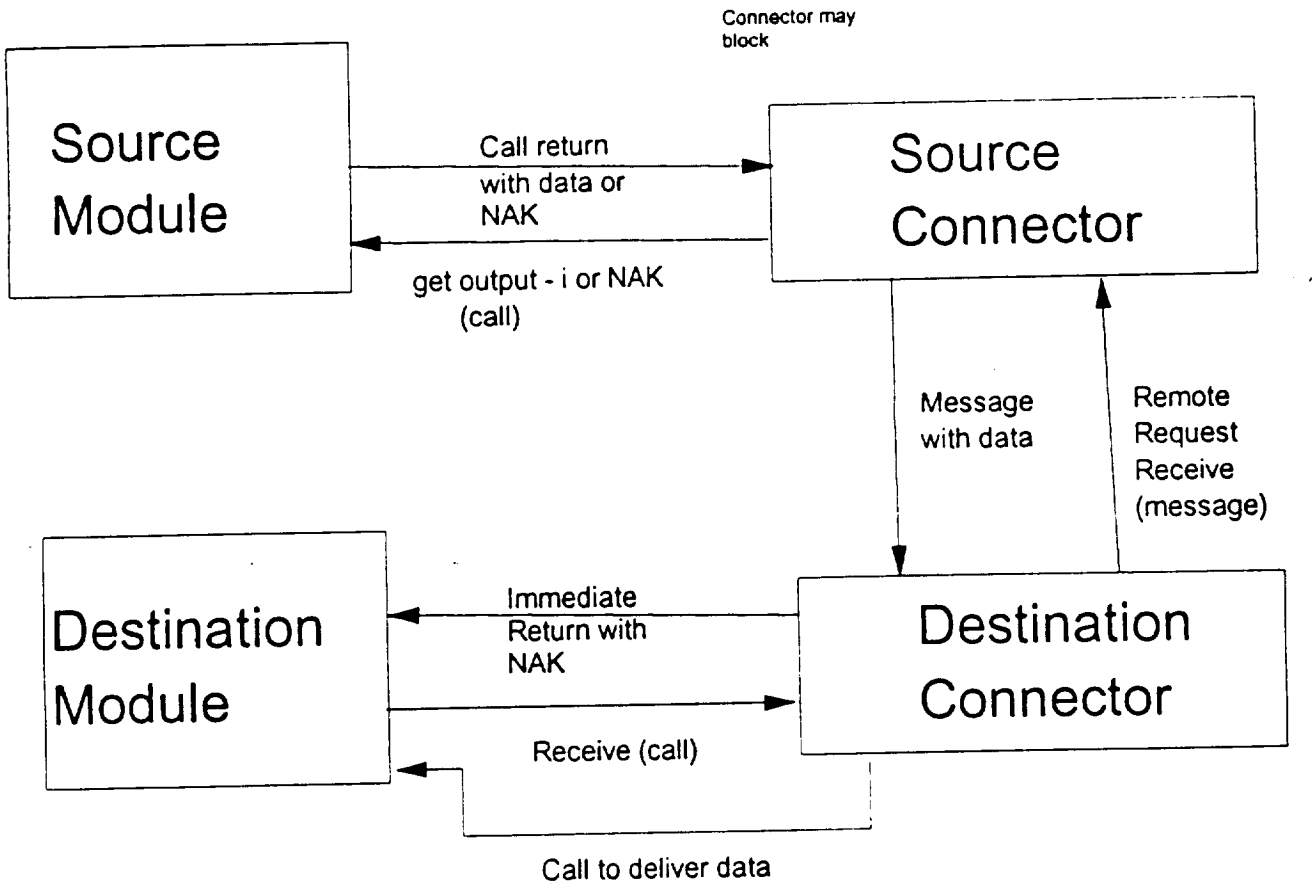


Figure 5.1

Principal Parallel
Computation Classes