NASA Contractor Report 177582

Modelling Parallel Programs and Multiprocessor Architectures with AXE

Jerry C. Yan and Charles E. Fineman

Contract Number NAS2-13210
May 1991

NASA
National Aeronautics and
Space Administration

NASA Contractor Report 177582

Modelling Parallel Programs and Multiprocessor Architectures with AXE

Jerry C. Yan and Charles E. Fineman
Sterling Federal Systems, Inc.
1121 San Antonio Road
Palo Alto, CA 94303

Prepared for
Ames Research Center
Contract Number NAS2-13210
May 1991

NASA
National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

Table of Contents

Figures and Tables

Summary

AXE, *An Experimentation Environment for Parallel Systems*, was designed to facilitate research for parallel systems at the "process level" using discrete-time simulation — without resorting to lengthy instruction-set level simulations or general stochastic models. It provides an integrated environment for specifying computation models, multiprocessor architectures, simulation, data collection and performance visualization. The user may study resource management strategies, parallel problem formulation, alternate hardware architectures, and operating system algorithms. AXE's simple, structured user-interface enables the user to model parallel programs/machines precisely and efficiently. Its quick turn-around time keeps the user interested and productive. The user can also observe the simulation on the color screen via four major panels:

- the *Multiprocessor Activity Panel* displays CPU bottlenecks and message routing;
- the *System Load Panel* illustrates the overall utilization of processing sites in the system;
- the *Global Traffic Panel* monitors the amount of inter-site communication; and
- the *Process Status Panel* traces the dynamic status of the software.

AXE models multiprocessors as a collection of *sites*. A *site* represents various operating system functions available for management and execution of the computation. The user may easily modify various architectural parameters of the multiprocessor such as the number of sites, connection topologies, routing algorithms, communication link bandwidths, and various CPU overhead for operating system activities.

Parallel computations in AXE are represented as collections of autonomous computing objects known as *players*. Players may be created either at compile-time or run-time. Each player is an instantiation of a specific *player type*. There are no shared data structures. Messages are exchanged asynchronously. A player may block to accept only messages of a specific *type* (while others queue up temporarily). Players may be used to represent parallel code-bodies (e.g. fork/join), communicating processes, Halstead's *futures*, remote procedures, and a subset of Hewitt's *actors*. A BEHAVIOR DESCRIPTION LANGUAGE (BDL) is used to specify such models. BDL program models preserve the interaction patterns between players and their processing requirements.

The AXE software package is being used at NASA Ames Research Center for research and development activities in high performance computing.

1. Introduction

1.1. Background

Realistic evaluation of new multiprocessor architectures and the accompanying resource management tools depends on the study of real applications on actual multiprocessor prototypes. However, the development of complete language and compiler tools, together with run-time environments is currently prohibitive for short term use in research. Therefore, *An Experimentation Environment*, AXE, was designed to facilitate such investigations at the "process level" using discrete-time simulation — without the need to resort to lengthy instruction-set level simulations or general stochastic models.

Researchers at Ames Research Center are using AXE to study dynamic load-balancing strategies for highly parallel systems [1]. It provides an integrated environment for computation model specification (using BDL), multiprocessor architecture specification, simulation, automated data collection and performance visualization. The researcher is able to study resource management strategies, parallel problem formulation, alternate hardware architectures, and operating system algorithms. AXE's simple, structured user-interface enables the experimenter to model different parallel programs and define various machines precisely and efficiently. Its quick turn-around time for experiments keeps the researcher interested and productive.

1.2. Software Architecture Overview

Three disciplines were observed when designing AXE:

1. structured user-interface — enabling the researcher to model different parallel programs and machines precisely and efficiently;
2. quick turn-around time for experiments — keeping the researcher interested and productive;
3. minimized need for re-compilation — AXE is structured so that a maximum number of parameters may be modified without needing re-compilation. These include machine characteristics, operating system parameters, input data, simulation data collection methods as well as the selection of heuristics for studying resource management strategies.

As shown in Figure 1-1, AXE is made up of two major components, the *Modelling Package* and the *Visualization Package*. The six components that make up both packages will be discussed in the next sections.

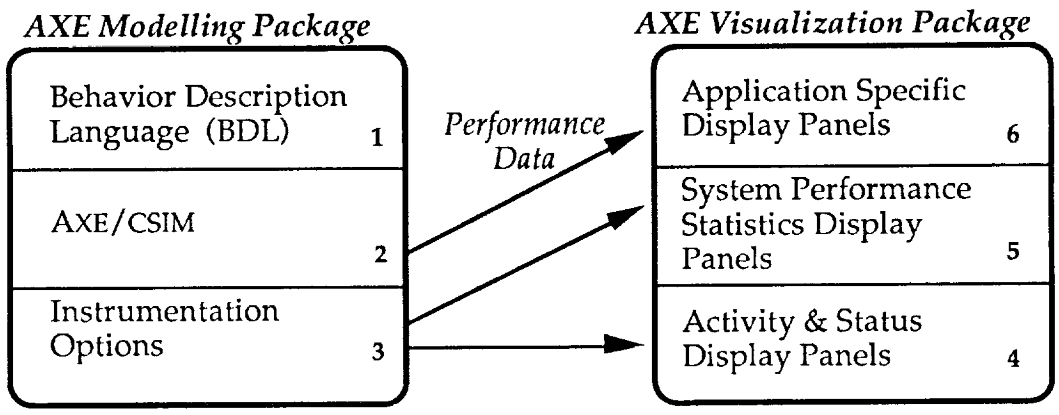

AXE Modelling Package
AXE Visualization Package
Behavior Description Language (BDL) 1
AXE/CSIM 2
Instrumentation Options 3
Performance Data
Application Specific Display Panels 6
System Performance Statistics Display Panels 5
Activity & Status Display Panels 4

Figure 1-1. Components of the AXE Experimentation Environment

1.2.1 The AXE Modelling Package

The *AXE Modelling Package* facilitates specification of parallel hardware and software models as well as simulation of their interactions. This package is made up of three basic software components as shown in Figure 1-1:

1. BEHAVIOR DESCRIPTION LANGUAGE (BDL) TRANSLATOR: Models of parallel computations are specified in AXE using BDL. The BDL translator, implemented as a separate front-end to AXE, converts BDL models into forms understood by other AXE modules. BDL follows an object-oriented LISP-like syntax. A parallel application may be specified using players that interact with one another via message passing. This communicating players paradigm gives the user a powerful and flexible tool for generating parallel application models whose performance is to be analyzed by AXE.
2. AXE/CSIM SIMULATOR: AXE/CSIM is a discrete-time event-driven simulator based on CSIM [2]. AXE/CSIM simulates the execution BDL program models on a variety of multiprocessor architectures. The performance behavior of various BDL program models running on various multiprocessor architectures may then be predicted and analyzed.
3. INSTRUMENTATION OPTIONS: The user may be interested in a variety of performance data which encompass application-specific issues, overall performance predication, load balancing information and many other performance issues. The Instrumentation Options component of AXE allows the user to define and generate the desired performance data sets to his/her particular interests.

1.2.2 The AXE Visualization Package

The data generated by the simulations by the *Modelling Package* are of little use unless the user has a clear and efficient means of displaying it. The *AXE Visualization Package* provides a flexible set of performance data analysis and representation tools. As shown in Figure 1-1, the *Visualization Package* may be classified into three major components.

4. ACTIVITY AND STATUS DISPLAY PANELS animate various performance parameters of the system during simulation. These displays can be used to examine the state of each processor and its interaction with other processors at every instant of the simulation. These "instantaneous" displays can be very useful in finding particular system bottlenecks.
5. SYSTEM PERFORMANCE STATISTICS DISPLAY PANELS plot the variation of performance parameters and statistics over a specified period of time. With these display panels, the user may identify the overall performance of the specified model. In addition, this class of display panels can be very helpful in identifying the various phase and state changes that may occur in the modelled system over time. Instead of having to monitor an animation of the system running over a long period moment-by-moment, the modeler may quickly recognize trends or patterns of performance that occur over time.
6. APPLICATION SPECIFIC DISPLAY PANELS: When evaluating a modelled system, it is often useful to correlate the system performance data with the behavior of the application that is causing the performance behavior. For this reason, the AXE Visualization Package allows the user to create display panels that are application specific.

1.2.3. AXE System Requirements

Portability has been a major consideration in the design and implementation of AXE. The workstation requirements for successful execution of AXE are as follows:

AXE Modelling Package runs on any hardware platform that CSIM executes (See [3]). These include Sun3, Sun4, SunSparc, DecStation 3100 and 5000, Sequent, VAXen etc.

AXE Visualization Package runs on workstation environments on which X-Windows (X11R4) are mounted. Examples include SGI, Sun3, Sun4, SunSparc, DecStation 3100 and 5000.

It should be noted that event trace files that are output by the *AXE Modelling Package* can become quite large. The user should take this consideration into account before attempting to run large complex models.

1.3. Report Outline

Chapter 2 describes the *AXE Performance Visualization Package*. Both hardware and software status are monitored. Examples of monitored parameters/activities include changing ready-queue lengths, message sending, CPU load distribution, and whether players are active, blocked, or idle. There are six major panels that can be displayed on one or two color displays. The user can pause during the playback of the simulation and reconfigure the screen to permit detailed observation of the activities displayed in any particular panel.

Chapter 3 describes how parallel architectures are modelled in AXE. Basically, AXE models multiprocessors as a collection of predefined *sites*. Machine topology and system overhead may be tailored further by the user parametrically. A discrete-time event-driven simulator based on CSIM is responsible for simulating the execution of BDL program models on these multiprocessor models.

Chapter 4 describes how parallel software is modelled in AXE. A parallel computation is represented as a collection of autonomous computing processes (or objects) known as *players*. Messages are exchanged asynchronously between players. A BEHAVIOR DESCRIPTION LANGUAGE (BDL) used to specify the behavior of players is also described. Two examples involving the building of abstract execution models for parallel programs are then given.

Chapter 5 summarized the work presented here and gives directions for future research and development.

2. The AXE Visualization Package

The use of color graphics to represent parallel program execution on multicomputers has been researched elsewhere to help interpret parallel system performance [4] as well as alternative architectures for executing parallel knowledge-based systems [5]. The monitoring facilities in AXE were designed primarily to display resource utilization in multicomputers and help locate software bottlenecks for player programs.

When compiled with the proper switches (see [6]), AXE will create a trace file containing events generated during a run. This event file is used to drive the *AXE Visualization Package* described in this section. Both hardware and software status are monitored. Examples of monitored parameters/activities include changing ready-queue lengths, message sending, player creation/termination, and whether players are *active, blocked, running* or *idle*. There are six major panels that can be displayed on one or two color screens. The user can pause during the playback of the simulation and reconfigure the screen to permit detailed observation of the activities displayed in any particular panel.

2.1. Display Philosophy

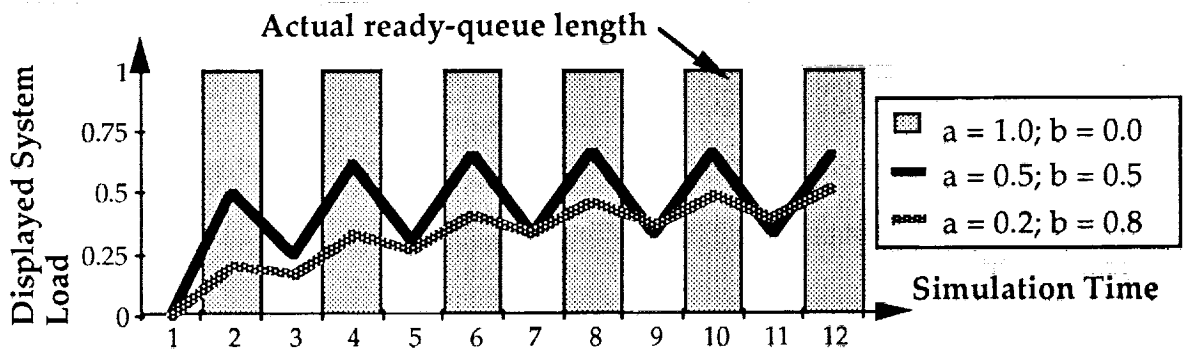


Figure 2-1. Effect of Displayed Load Values Using Various Values of *a* and *b*

Before discussing what the color bars that change heights mean in each panel, the relationship between the actual color (and height) observed and the actual parameter value are presented here. Instead of using the instantaneous lengths of the ready-queue and message queues directly, a *moving-average* function is used with the AXE Visualization Package. Suppose that the length of the ready-queue (at some site) at time τ is λ_τ. If the display is to be updated at time τ_1, the *displayed queue-length* is computed according to $a\lambda_{\tau_1} + b\lambda_{\tau_2}$ where λ_{τ_2} = the last displayed value, a and b are non-negative constants and $a + b = 1$. The *moving-average* function is used to filter out rapid varying load values so that the researcher can observe general trends in load variation easily. As illustrated in Figure 2-1, in order to track the actual load more closely, higher values should be chosen for a .

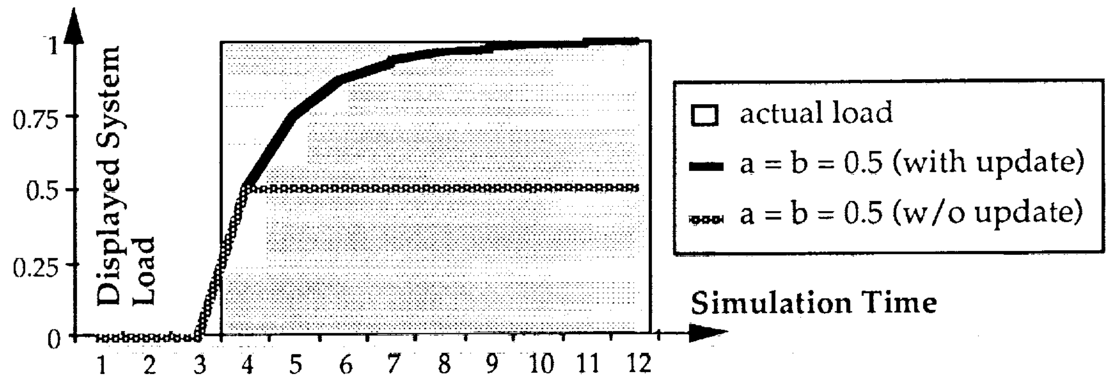


Figure 2-2. Effect of a "Self-updating" Panel

Because AXE is an event-driven simulator, the panel is updated (by default) only every time a process is added or removed from the ready-queue. Unfortunately, when a process is scheduled and executes for a long time, the load value at that site will never reach the actual value because of the effect of the moving-average function. Figure 2-2 illustrates this situation when the load of a site is changed from 0 to 1. Therefore, the ACTIVITY AND STATUS DISPLAY PANELS are designed so that each site updates its load value every now and then. This update interval is settable by the user (see [6]).

2.2. Panel Description

2.2.1 Multiprocessor Activity Panel

The Multiprocessor Activity Panel displays CPU usage and message routing. As shown in Figure 2-3, a multicomputer which consists of 16 processing sites connected as a 2-dimensional grid is simulated. A line joining two adjacent sites indicates that a message is being transmitted between them. A vertical color bar on the left-hand-side of the panel illustrates the color scheme used — light blue representing the lowest load, through green, yellow and red, to magenta as the highest. Each site contains two color columns. The one on the left-hand-side (called *CPU load indicator*) indicates how busy the CPU at that site is by both changes in height and color. The one on the right hand-side (called *router processor load indicator*) illustrates how busy the communication processor is using a similar color scheme. Where the color of the *CPU load indicator* represents average length of ready-queue, the color of the *router processor load indicator* indicates the average number of packets waiting to be routed at each site. The relative width of *CPU load indicator* and the *router processor load indicator* can be adjusted to display the distribution of CPU or traffic hot spots.

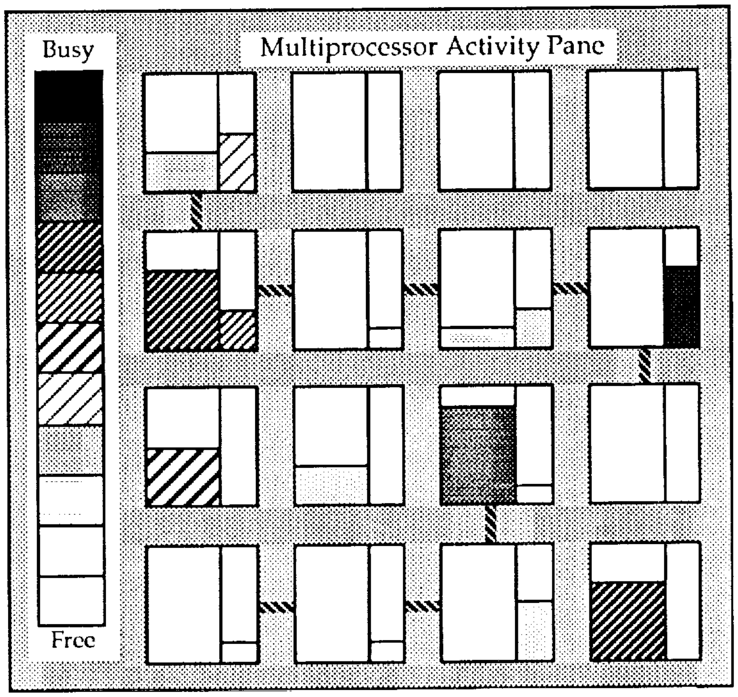


Figure 2-3. The Multiprocessor Activity Panel

2.2.2 System Load Panel and Global Traffic Panel

The System Load Panel and Global Traffic Panel look very much alike as Figure 2-4. illustrates. The System Load Panel displays the overall utilization of processing sites in the system. The number of active sites is plotted against simulation time. The Global Traffic Panel, on the other hand, monitors the amount of inter-site communication. The rate of messages sent across sites is plotted against simulation time. Data of more than one simulation can be plotted on top of one another. This feature is especially useful when comparing the performance of two mappings.

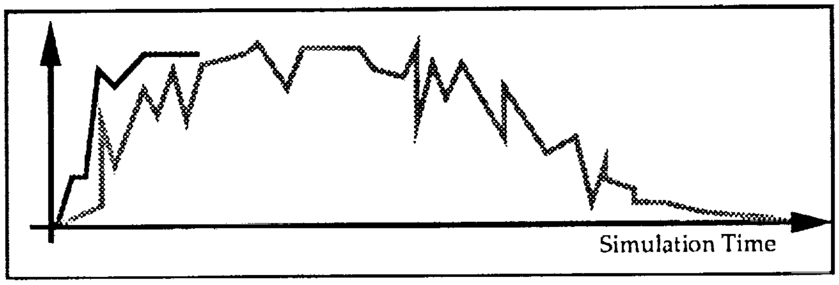
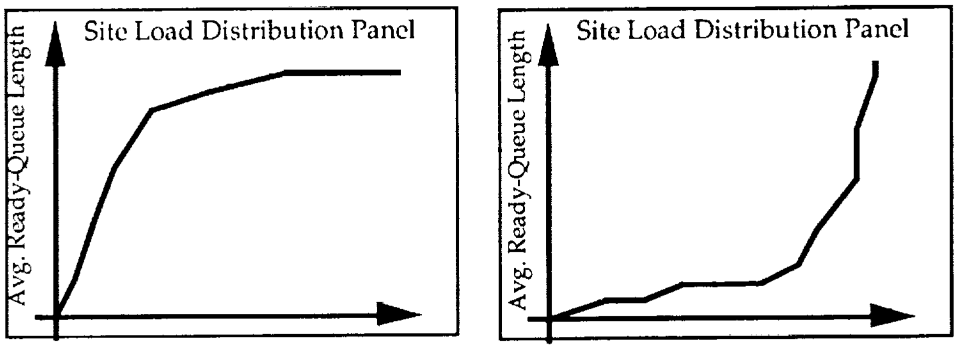


Figure 2-4. The System Load Panel and Global Traffic Panel

2.2.3 Site Load Distribution Panel

The Site Load Distribution Panel plots the "*load*" of all the sites in ascending order. The *load* of a site is the average ready-queue length for some user-settable time interval. Load distribution is "even" if the curve resembles a "plateau" as shown in Figure 2-5a.



(a) "Closer-to Ideal" Distribution **(b) Undesirable Distribution**

Figure 2-5. The Site Load Distribution Panel

2.2.4 Process Status Panel

The Process Status Panel (Figure 2-6.) traces the dynamic status of the software. Like the multi-processor activity panel, the 2-dimensional grid of sites is represented as an array of rectangles.

Inside each site are a number of smaller rectangles, each of which, in turn, is divided into two color columns. The one on the left (called the "message queue-length indicator") indicates the number of unprocessed messages queued up in the message buffer of that particular player. The one on the right, called "player status indicator", represents the state each resident player is in:

Empty (☐): no player is instantiated;

Yellow (▧): player is *scheduled*, ready to make use of (but have not got hold of) the CPU (i.e. it is in the *ready queue*);

Red (▨): player gets hold of the CPU and is *executing* (under the currently available models, there will be at most one such player per site);

Blue (⁘): player is *blocked*, waiting for the arrival of a specific message from another player (e.g. the *reply* to a message it sent earlier, the *evaluation* of a *future* or the *return* of a *remote procedure call*);

Grey (▒): otherwise, the player is in a *dormant* or *idle* state, waiting to be *activated* by the arrival of a message.

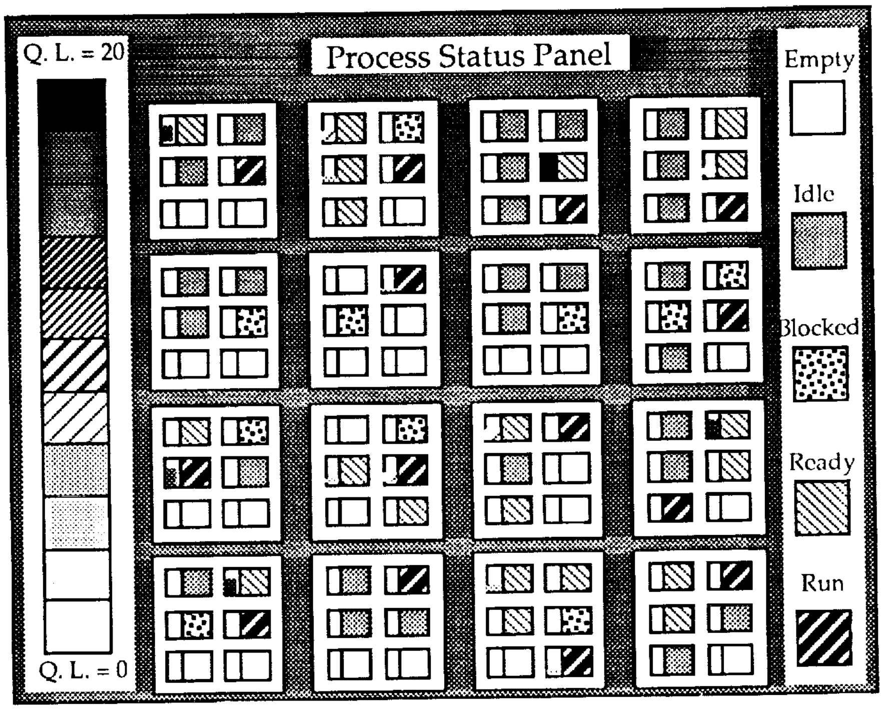


Figure 2-6. The Process Status Panel

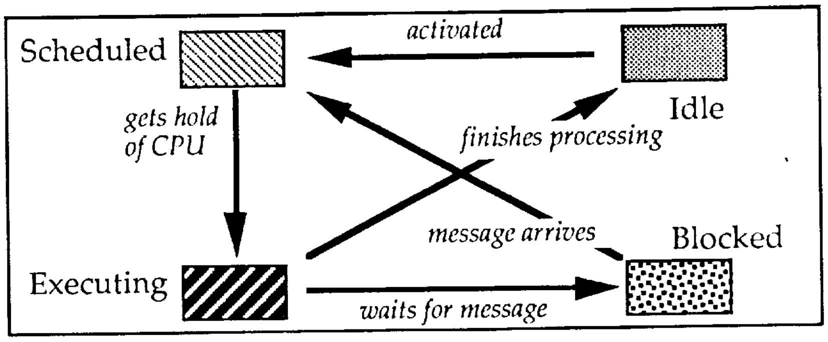


Figure 2-7. State Transition for a Computing Process

In the case of Figure 2-6, up to 6 players can be monitored at each site. Figure 2-7. illustrates the events that cause state transition for processes. The relative width of the message queue-length indicator and the player status indicator can also be changed. Software bottlenecks can be located by spotting players with long message queues. Figures 2-8 on the next pate illustrates how all the above mentioned panels will fit together on a screen.

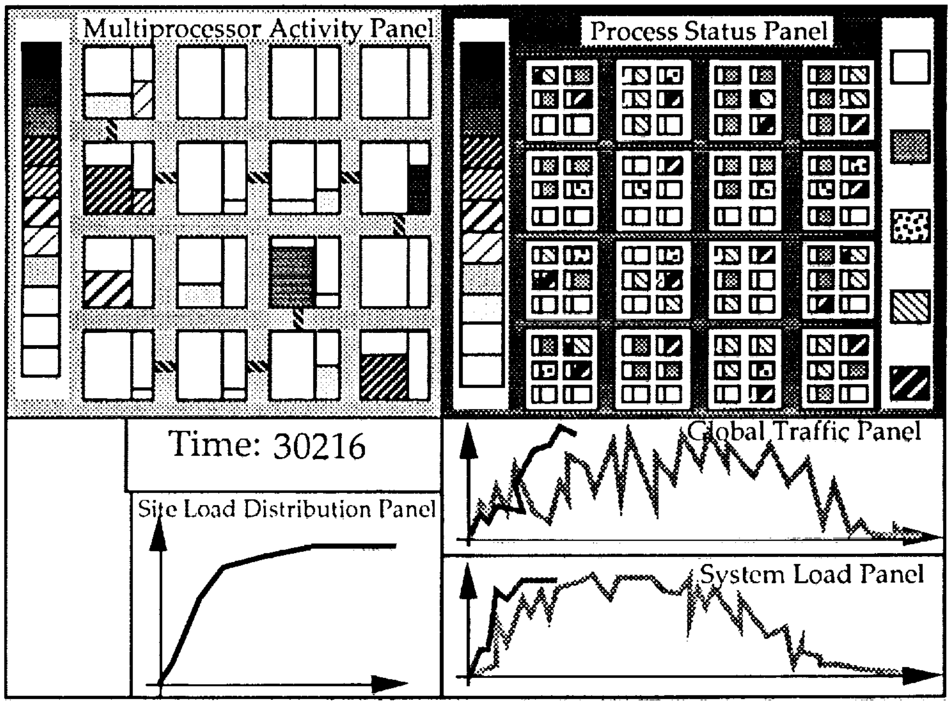


Figure 2-8. Screen Layout for Grid Multiprocessors

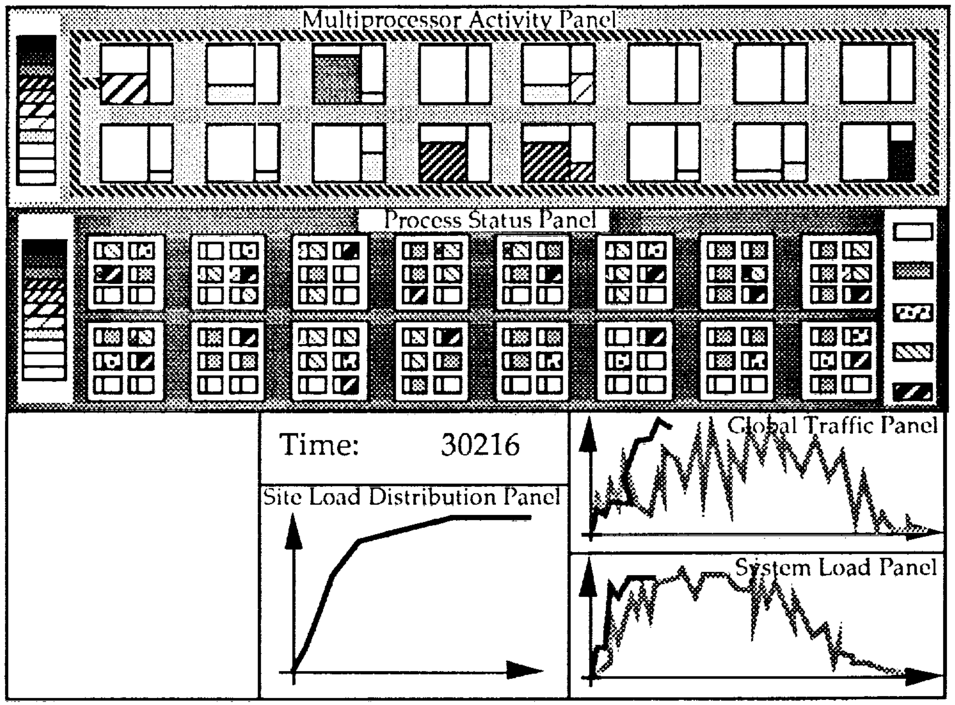


Figure 2-9. Screen Layout for Distributed Systems on a Token-Ring

2.3. Controlling Instrumentation/Visualization

2.3.1 Displaying Multiprocessors of Different Topologies

Depending on the topology of the simulated multiprocessor, the Multiprocessor Activity Panel and Process Status Panel will be displayed differently. Figure 2-9 illustrates how a token-ring connected multiprocessor will be displayed on the screen. The basic operations of the panels are identical. A line segment joining a site and the ring indicates that site currently holds the token, and is sending a packet out onto the ring.

2.3.2 Display on Multiple Screens

The AXE Visualization Package can drive multiple displays on multiple workstations connected to the same network through the X-Window protocols. When two displays are available, the user may wish to develop an *Application Specific Panel* to help visualize the state and progress of the application in more *real world* terms. For example, Figure 2-10 illustrates the display of a real-time aircraft tracking system ELINT [7] along side a standard AXE display. Here, we can spot where the aircrafts are currently located ($\overset{12}{□}$) and where the real-time software *thinks* the aircrafts are ($\overset{12}{⊠}$) during simulation. It should also be noted that the System Load Panel, the Global Traffic Panel and the Site Load Distribution Panels are displayed on the second screen, balancing the amount of information presented on both screens.

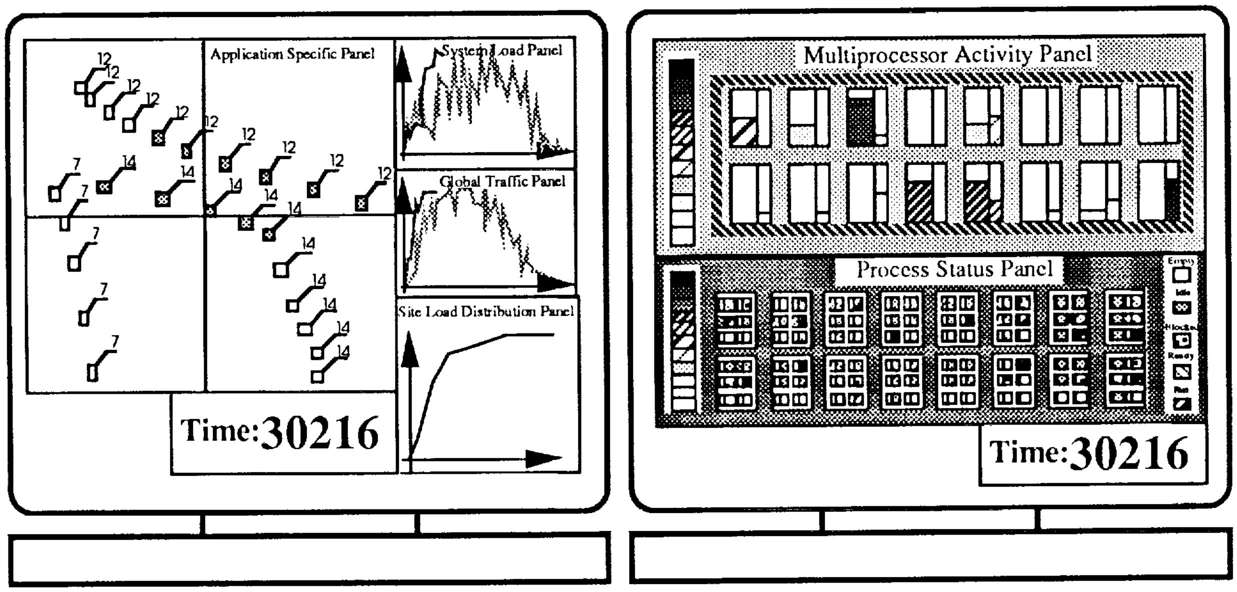

Application Specific Panel
System Load Panel
Global Traffic Panel
Site Load Distribution Panel
Time:30216
Multiprocessor Activity Panel
Process Status Panel
Time:30216

Figure 2-10. Multiple Screen Layout for Token-Ring Systems

3. Multiprocessor Hardware Models

3.1. Functional Model of a Site

AXE is designed to model *multicomputers*; they consist of homogeneous processing elements (*sites*) connected in a nearest neighbor fashion by physical communication links. Each site is autonomous — with its own local memory, processor unit and an operating system kernel governing message forwarding, task scheduling, as well as memory management. The local memory of each site is assumed to be large enough to hold any number of players. CPU time is allocated to players on a FIRST-COME-FIRST-SERVE basis (FCFS) by default at each site. If the CPU is busy when a player makes a CPU request, the requesting player will be put on a *ready-queue* until the CPU becomes available. Players may still be preempted by the *mailer processes* at each site responsible for message routing. AXE also implements a ROUND-ROBIN scheduling algorithm and allows more than one player at a site to process concurrently. The quantum size is settable by the user but the simulator has to be recompiled.

3.2. Building Multicomputer Models with User-Settable Parameters

3.2.1 Parameters Controlling Connection Topologies

A "machine configuration file" defines the topology and size of the multicomputer to be simulated. Changes to this file will only take effect when the simulator is recompiled. Table 3-1 summarizes how different multicomputers can be defined by setting constants to different values.

Topology	Number of sites	Constant Definitions
Token ring	n	#define TOKNRING
Grid	n_x by n_y	#define GRID #define DIM_X n_x #define DIM_Y n_y #define NO_OF_SITE $(n_x * n_y)$
Hypercube	2^n	#define IPSC2 #define NO_OF_DIM n #define NO_OF_SITE (2^n)

Table 3-1. Parameters Controlling Connection Topologies

3.2.2 CPU and Routing Overhead

There were several timing parameters used in the simulation. The overhead incurred by these activities are described in terms of 5 user settable parameters as shown in Table 3-2: T_{hop}, T_{token}, T_{tokgen}, $T_{deliver}$, and T_{place}. For hypercubes, messages routing time depends on message lengths. The default transmit rate is set at 3.07ms/kbyte; this, of course, can be changed easily.

3.3. Message Passing Protocols

Message routing is implemented by two (CSIM) processes, a *local mailer* and *remote mailer* at each site. The *local mailer* is responsible for helping resident players send out messages. The *remote mailer* is responsible for message forwarding as well as delivering messages received from other sites to resident players.

RECEIVING MESSAGES — Whenever a player executes the statement (**WAIT** `some_type`), several possible scenarios exist.

- If a message of the specified type has not arrived for the calling player, the player process will block (waiting for the message).
- If a message of the specified type has arrived for the calling player, the player is placed on the ready queue.
- If more than one such message exists, the first one received will be accepted.

Parameter	Token-Ring Model (ms)	Multicomputer Model (ms)
T_{hop}	time needed for a site to place a packet onto the ring = 0.064	time needed for sending a packet to a neighboring site = 0.73
T_{token}	time needed for a bit to travel from one site to the next[1] = 0.01	N/A
T_{tokgen}	time needed for a site to place the token onto the ring = 0.01	N/A
$T_{deliver}$	CPU overhead to remove a message from the ring = 0.144	time needed to send a packet to a local player = 0.01
T_{place}	CPU overhead for process creation = 0.25	

Table 3-2. Timing Parameters for Routing and Other Overhead

[1] Note that the token circulation time when no messages are being sent is (T_{token} * NO_OF_SITE).

SENDING MESSAGES — The *local mailer* at the sender site first decodes the *mail-address* of the receiver to decide where it resides (i.e. locally in same site in which the sender resides or remotely on a different site):

- If the receiving player resides locally (e.g. in Figure 3-1, player C at site 2 sends player D a message at site 2) there are still three possible cases:
  - i) If the receiving player is either idle or blocked waiting for this message, the message will be delivered by the *local mailer* and the receiver placed on the ready-queue.
  - ii) If the receiving player is running, the *local mailer* delivers the message to the player's message buffer. When the player finishes processing, it will check this message buffer and find the message.
  - iii) Otherwise, the receiving player is expecting another message (type); this message will be put on the receiver's message buffer and the receiver remains blocked.
- If the receiving player resides in another site (e.g. in Figure 3-1, player A at site 1 sends player D at site 2 a message): The *local mailer* decides that the receiver does not reside locally (based on the receiver's *mail-address*) and delivers the message to the *remote mailer*. The *remote mailer* then invokes a routing algorithm to determine the immediate neighbor through which this message should be routed and sends the message to the *remote mailer* at the selected neighbor. When the message finally arrives at the receiver site, it is delivered directly (by the *remote mailer* of that site) to the receiving player.

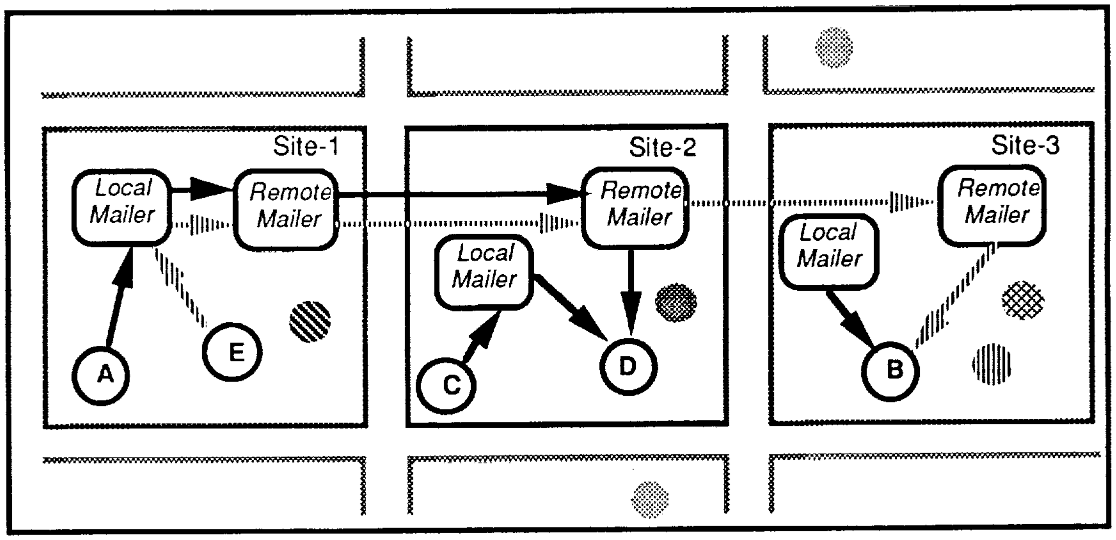


Figure 3-1. Message Passing on Multicomputer Models Simulated in AXE

FORWARDING MESSAGES — The *remote mailer* is also responsible for forwarding messages (e.g. in Figure 3-1, the *remote mailer* at site 2 forwards a message sent by player E at site 1 to player B at site 3). The simplest class of routing strategies on multicomputers is known as store-

and-forward. Messages *hop* from site to site until they reach their destination. Routing target selection may be static or dynamic. Dynamic routing algorithms select message routes based on the (static) topology of the communication network as well as the amount of traffic in specific parts of the system. AXE provides a variety of message forwarding schemes depending on the chosen architecture. These schemes are described in the following sections.

3.4. "Diagonal-First" *Store-and-forward* Algorithm

A simple static diagonal routing strategy is used in AXE for an *n-nary m-cube* type connection topologies (for a total of m^n sites) to implement *store-and-forward* routing:

- The address of a site s_x is represented by an ordered m-tuple $(x_1, x_2, \dots x_m)$ — where $0 \leq x_i \leq n$;
- Suppose a packet is to be sent from site s_{x_0}, to s_{x_d}
  - The address of the *site of origin* (s_{x_0}) is represented as $(x_{o1}, x_{o2}, \dots x_{om})$.
  - The address of the *site of destination* (s_{x_d}) is represented as $(x_{d1}, x_{d2}, \dots x_{dm})$.
- Define a direction vector $\Delta = (x_{\delta 1}, x_{\delta 2}, \dots x_{\delta m})$ where $x_{\delta i} = x_{di} - x_{oi}$
- If $\forall_i \; x_{\delta i} = 0$, then $s_{x_0} \equiv s_{x_d}$, i.e., the *site of destination* is *the site of origin*
- Otherwise, let t = position of $\max(|x_{\delta 1}|, |x_{\delta 2}|, \dots, |x_{\delta m}|)$
- if $x_{dt} > x_{ot}$, then the neighbor site selected is $(x_{o1}, \dots, x_{ot}+1, \dots x_{om})$
- if $x_{dt} < x_{ot}$, then the neighbor site selected is $(x_{o1}, \dots, x_{ot}-1, \dots x_{om})$

3.5. Message Sending on Hypercube Models

Besides store and forward networks, users may choose to simulate their application using a model based on the Intel iPSC/2 Hypercube. As part of this model, we have implemented a version of *worm-hole* routing [8, 9]. The basic idea behind this type of routing is that a hardware channel is set up between the two communicating sites that remains established until the entire message get transmitted. Here is a scenario to illustrate what happens in the system when a message is sent:

1. Source site (s_s) sends a request for connection (REQ) to destination site (s_d).
2. Intermediate sites receive REQ; routing logic decides who the next target site is and which output channel to be used.
3. Allocate the output channel if it is free; otherwise, REQ is blocked inside the routing logic until the required channel is available.
4. s_d receives the REQ and sends an ACK back to s_s.
5. s_s starts to pump the message to s_d.

6. EOM (end of message) is appended at the end of the message. Each site releases the corresponding output channel immediately after it receives EOM.

As an example, Source node 1010 wants to send a message to Destination node 1111. The route the message takes is unique: 1010 → 1011 → 1111. The reason is:

1. Compute the "*probe*" = *exclusive-or* of s_s's ID and s_d's ID is = 0101
2. s_s sends message via channel 0 to node 1011 because the least significant 1-bit is bit 0.
3. Compute the "*probe*" = *exclusive-or* of 1011 and s_d's ID is = 0100
4. Because the least significant 1-bit is bit 2, site 1011 uses channel 2 to send message to node 1111.

3.6. Message Sending on Token-Ring Models

In addition to the above models, AXE is also able to model homogeneous distributed systems connected via a token ring. A single token ring with a solitary rotating token is modelled. A site on the ring needs to obtain the token for each message it sends. The token was regenerated by a site immediately after the message was sent. This method is more efficient than allowing the message to travel once around the ring before regenerating the token. This allowed for the possibility of having multiple messages on the ring at any one time. For example, consider a token ring system of three sites: *A*, *B*, and *C*. A token travelling around the ring goes from *A* to *B* to *C* before returning again to *A*. Suppose A wants to send a message to *C* and *B* wants to send a message to *A* and the token is just about to pass *A*. *A* grabs the token and sends its message. Immediately after sending, it regenerates the token. *B* obtains the token and sends its message before the first message reaches *C*. Thus, two messages are on the ring simultaneously. With this configuration and the right timing values, if there are *n* sites on the ring, it is possible to have n messages on the ring simultaneously. Future versions of AXE could include more elaborate and realistic protocols such as bi-directional rings, multiple message frames with the appropriate protocols, dual-redundant and counter-rotation configurations etc..

4. Parallel Software Models

4.1. The Players Programming Paradigm

A parallel computation is represented as a collection of autonomous computing agents known as *players*. Players may be created either at compile-time or run-time. Each player is an instantiation of a specific *player type* — which characterizes i.) the structure (i.e. slot-names) or its internal state values, ii.) the messages it understands and iii.) how it responds to different message types. A player's internal state values are only accessible by the player itself. Each state value contains either a numerical value or a *mail address* of (or "a reference to") some player.

There are no shared data structures. Messages are exchanged asynchronously. In other words, the sending player does not need to suspend computation and wait for an acknowledgement from the receiver. By default, messages are processed in the order in which they are received. A player has the option to accept only those messages under a specific *message type* (while all the others queue up temporarily).

During program execution, a player is either ACTIVATED, RUNNING, BLOCKED or IDLE. These four states are defined as follows:

- ACTIVATED — player is waiting in the ready-queue (ready to make use of the processor);
- RUNNING — player is using the processor;
- BLOCKED — player is waiting for the arrival of a specific message (or) from another specific player (e.g. the "reply" to a message it sent earlier, the "evaluation" of a *future* [10] or the "return" of a *remote procedure call* [11]);
- IDLE — player is waiting to be *activated* by the arrival of a message.

In response to messages received, a player may create other players, modify its internal states, send messages or check and wait for messages of a particular type. An activated player selects the appropriate handling procedure in response to the message. These handling procedures are defined as part of the *player-type definition.*

Player programs can be described using a PARALLEL PROGRAM BEHAVIOR DESCRIPTION LANGUAGE (BDL). Activities such as message sending/blocking, player creation as well as computation can be described using BDL. BDL follows a LISP-like syntax. Its semantics and syntax are informally introduced here. A more detailed description of the language is given in the next section.instantiation

Figure 4-1. shows a simple set of player constructs and interactions. We see two basic constructs that are crucial to the understanding of the Players Programming Paradigm. A segment of program text is shown on the left; the corresponding computing agents (i.e. players) are shown on the right. Some important points to take note of in this example are:

1. The PLAYER-TYPE DEFINITION defines the player's internal state variables that will be used during the execution of the model, messages it understand as well as how it responds to incoming messages.
2. Players communicate via message passing with BDL commands such as **post** and **wait**.
3. Globally known players are created using the PLAYER-INSTANCE DECLARATIONs.

4. Each player contains its own copy of its internal state variable which are not directly accessible by other players.

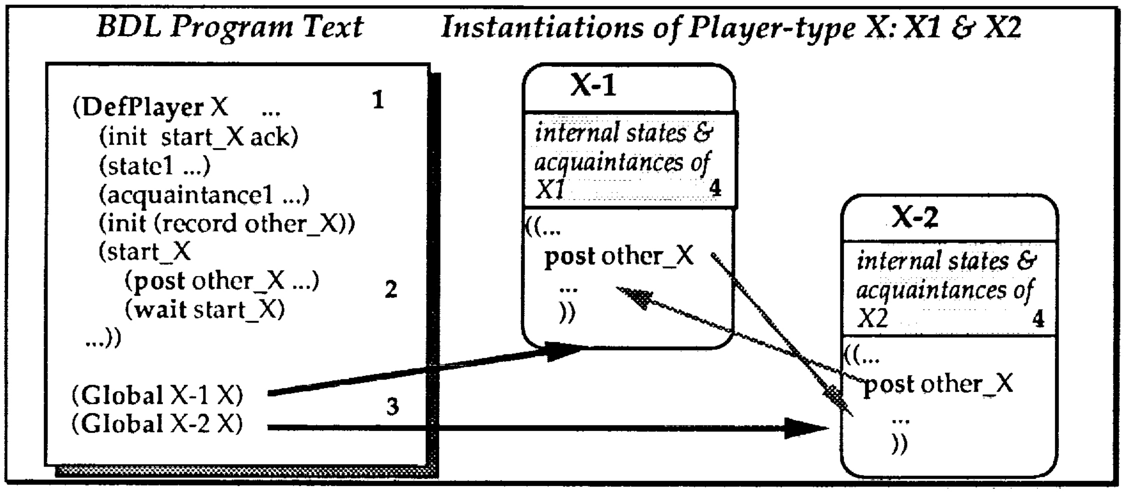


Figure 4-1. A Simple Diagram of Players and Their Interactions

4.2. BDL — A Behavior Description Language for Parallel Programs

The following conventions will be adhered to when presenting different language constructs in this section:

- Items in boldface appear verbatim (but case insensitive)
- a term followed by a "*" appears zero or more times
- a term followed by a "+" appears one or more times
- a term followed by a "?" is optional
- two terms separated by a " | " are alternatives
- <identifier> refers to an alphanumeric string (underscores permitted) that does *not* begin with a number.
- <number> refers to an integer.
- <c-code> refers to a segment of C program text delimited by braces "{" and "}".

4.2.1 Top-level Constructs

There are two types of top-level constructs in the BDL language: *player-instance declarations* and *player-type definitions*. *Player-instance declaration* (**Global**) instantiates a player at compile time. *Player-type definition* (**DefPlayer**) defines the internal structure of a player type and its message handling procedures.

The mail-address of a player created at compile-time is "globally known". Players created at run-time can send messages to or wait for messages from these "globally known" players using its *name*. The **Global** construct instantiates a player at compile time:

```
( Global <identifier> <identifier> <location> )
where <location> ::= ( id <number> ) | ( xy <number> <number> )
```

The first identifier is the *name* of the player instance. The second is the *type* of the player. The location clause requires the programmer to specify the site at which the player is to be placed. The current implementation of AXE supports two specification methods:

id — an integer from 0 to (NO_OF_SITE -1) specifies the site ID

xy — When a 2D Grid is used, *Site[x,y]* may be specified directly

The **DefPlayer** construct defines the internal structure of a player type and its message handling procedures:

```
( DefPlayer <identifier>
  ( <identifier>* )
  ( <decl>* )
  ( <decl>* )
  <c-code>?
  ( <behavior> )+
)
where,  <decl>      ::= <identifier>  |  ( <identifier> <number> )
        <behavior> ::= <identifier> <statement>+
```

The *player-type definition* can be divided into six parts:

- The first <identifier> indicates the name of the player type
- The first list defines the *message types* recognized by this type of player. These names will either appear as the first identifier of the *behavior clauses* of this player definition or be used by **Wait** statements later.
- The second list defines the *acquaintances* of the player. An *acquaintance* holds the *mail address* of a player. *Acquaintances* can either be set by creating new players (using the **Make** construct) or by receiving *mail addresses* of created players from others (using the **Record** construct).
- The third list defines the *state variables* of the player. They may be set and used as integer values.
- Following the lists of definitions comes the preamble. This is an optional segment of C code that contains support for the player that could not be coded using the standard BDL language features. This sequence of C statements may contain data declarations as well as executable

code. It will be inserted into the player's generated code just before the main message processing loop.

- The last part of the player definition is the list of *behavior clauses*. Each *behavior clause* comes with a *head* — indicating the corresponding *message type* and a *tail* — a list of executable statements. When an idle player receives a message, it checks the in-coming message's type against the *head* of each *behavior clause*. If it finds a match, the *tail* will be executed. If not, the message will remain in the buffer and the player remains idle until a message whose type matches the *head* of a *behavior clause* arrives. That message may still be extracted should the player execute the **Wait** statement.

Both *acquaintances* and the *state variables* may be declared as scalars or vectors (1-dimensional arrays). Individual elements of the vector may be referred to by the aref construct. Array indices begin at 0. The user may also insert top-level C code into the file generated by BDL. This code would likely include data structures and routines that cannot be defined using standard BDL constructs.

4.2.2 Player Creation

The **Make** construct creates a player at run-time:

```
( Make <placement>?   ( ( <identifier> |  * )  <identifier> )+ )
where <placement> ::=      :any
                     |     :by_load
                     |     :from_file
                     |     :at   <location>
                     |     :default
```

The first parameter specifies the placement scheme for this creation. It directs the operating system at the creat<u>or</u> site how to choose the site where the creat<u>ed</u> player will reside. The current version of AXE supports several placement schemes:

- The simplest placement scheme is **any** — players are created at randomly chosen sites.
- The **by_load** scheme creates a player at the site with the fewest resident players. In case of ties, the site with the smallest site ID is chosen.
- The **from_file** scheme reads the site ID's to be used from a text file. The name of this file is given in the *configuration file* (see [6]). Upon seeing an end-of-file, AXE begins reading at the start of the file again.
- The **at** scheme allows the user to specify the resident site of the new player explicitly. The location is specified as with the **Global** construct except that expressions (as opposed to numbers) are also permitted when specifying sites.

• The **default** scheme instructs AXE to check the *configuration file* for the scheme to be used (see [6]).

The remaining parameters are pairs of slots and player types. The first value of each pair designates a location for the *mail address* of the newly created player. If a "*" is given for this location, the address of the newly created player is thrown away. If an identifier is given, it is assumed to be either one of the creator player's *acquaintances* or a *global player name*.

4.2.3 Data Manipulation

BDL provides support for accessing, modifying, and computing other values with local variables. As stated above, BDL supports vectors (1-dimensional arrays). Individual elements of these arrays may be accessed using the **Aref** construct:

(**Aref** <identifier> <expression>)

Scalar values may be set using the **Setq** construct:

(**Setq** <lhs> <expression>)
where <lhs> :: = (**Aref** <identifier> <expression>) | <identifier>

In addition, the value of a scalar may be modified using the **Dec** and **Inc** constructs:

((**Dec** | **Inc**) <lhs>)

They decrement and increment their argument respectively.

<expression> refers to either a scalar value (either an *integer constant* or an *internal state variable*) or a compound expression built using the BDL operators. In addition, a C code segment surrounded by braces may be used anywhere an expression is called for. In this way, users may perform calculations that are not available within standard BDL. BDL provides the basic arithmetic operators (+, -, *, /) as follows:

(<operator> (<expression>)+)

If only one argument is passed, the results for "-" and "/" are the negative and reciprocal of the argument respectively.

4.2.4 Message Passing

The **Post** construct is used for message sending:

(**Post** <acquaintance> <message-type> <message-term>*)
where, <message-term> ::= **self**
| <acquaintance>
| <expression>

```
                                |  ( <identifier> <expression> <expression> )
                                |  :length <expression>
```

The <acquaintance> term refers to the mail address of the receiving player. The <message-type> term specifies the message handler procedure to be used by the receiver. The body of each message may be made up of any combination of message terms:

- **self** — the *mail address* of the sending player;
- <acquaintance> — a *mail address* of a player;
- <expression> — the *resulting integer* value of the expression;
- **(** <identifier> <expression> <expression> **)** — portions of arrays; the <identifier> indicates the array; the second argument refers to the starting element; the third refers to the number of elements to be passed;
- **:length** — this form allows the length of the messages to be specified. By default, each message is of length 1 (Byte).

The **Reply** construct is similar to the **Post** construct except that no receiving player (i.e. <acquaintance>) has to be specified. Suppose player ***b*** sends a message to player ***a*** and ***a*** now executes a **Reply** statement:

i) if no **Wait** statements have been executed before the **Reply** statement, ***a*** replies to ***b;***

ii) otherwise, ***a*** replies to the sender of the message which ***a*** accepted with the **Wait** statement most recently executed.

The syntax of the **Reply** statement is as follows:

```
(Reply <message-type> <message-term>*  )
```

Besides receiving messages when idle, players may wait for messages within the body of a *behavior clause*:

```
( ( Wait | Receive ) <message-type> ( :from <acquaintance> )? )
```

If a message of the specified type has not arrived, the executing player will block. The **:from** key-word allows the receiver to wait for a message of a certain type to arrive from a particular player. Executing the **Wait** statement will change the player to whom the executing player replies whereas **Receive** will not.

The **Record** construct extract the arguments associated with the last message received into the corresponding states:

```
(Record (  <acquaintance>
           | <lhs>
           | (<identifier>  <expression>  <expression> ) )+ )
```

Data extraction is done on a term-by-term basis; terms in the **Record** are matched with the corresponding term in the **Post** that sent the message. If the receiver executes **Record** with fewer arguments than the sender does with **Post**, these extra values will be dropped. On the contrary, if **Record** is called with too many arguments, the extras will be filled with zeros.

If a scalar is sent but a vector is expected, the first element of the vector is set to the scalar received and the remaining elements are set to zero. If a vector is sent but and a scalar expected, the scalar is set to the first element of the vector. If the vector sent is longer than the receiving vector, as many elements as possible are recorded (without going beyond the bound specified in the **Record**). If there are too few, the remaining elements are set to zero.

4.2.5 Flow Control

The **If** statement evaluates a boolean expression and executes one of two alternatives depending on the value of the expression:

```
( If <boolean-expression> <statement> <statement>? )
  where, <boolean-expression> ::= <expression>
                                | ( <relop>  <expression> <expression>  )
                                | ( or  <boolean-expression>+  )
                                | ( and  <boolean-expression>+  )
     <relop> ::= <  |  <=  |  ==  |  <>  |  >=  |  >
```

If the boolean expression evaluates to true (or non-zero), the first statement is executed, otherwise the second one (if present) is executed. Multiple statements may be encapsulated using the **Progn** construct.

The **Repeat** construct defines a loop to be executed a specified number of times:

```
(Repeat <expression>  <statement>+ )
```

Branch statements allow probabilistic behavior to be modelled:

```
( Branch ( ( <expression>  <statement>+ ) )+ )
```

The **Branch** statement contains a list of clauses, each of which has a specified probability of occurring. For example the following statement indicates that the first branch is taken with a probability of 40%, the next one 50% etc..

```
(Branch  (40           (... ) (... )  ... )
         ((+ 20 30)    (... ) (... )  ... )
         (10           (... ) (... )  ... ))
```

The current implementation does not check that the probabilities of all the branches add up to 100%. Consider the following statement: (**Branch** (v_1 (...) (...) ...) (v_2 (...) (...) ...) (v_3 (...) (...) ...)). A random number (r_n) between 1 and 100 is selected. The first branch will be taken if $r_n \le v_1$; the second branch will be taken if $v_1 < r_n \le (v_1 + v_2)$, and so on.

4.2.6 Miscellaneous

The **Run** construct requests to use the processor for a specified duration. It should also be noted that these "time units" are meaningful only when discussed in conjunction with other characteristics of the hardware (such as communication link bandwidth and context-switch overhead etc.). It's syntax is:

(Run <expressions> **)**

The **Hold** construct instructs AXE to advance the specified amount of simulation time without requesting the use of a processor. In other words, the caller player *pauses* for a specified amount of time before executing the next statement. It's syntax is:

(Hold <expression> **)**

The **Funcall** construct is used to call predefined C routines. Integer arguments are listed after the function name:

(Funcall <function-name> <expression>* **)**

The **Progn** construct allows multiple statements to be parsed as a single statement. It is particularly useful with the **If** statement discussed earlier:

(Progn <statement>+ **)**

The **Destroy** construct terminates the caller player process: **(Destroy)**

The **Genesis** construct marks the beginning of the simulation. All events that occur before **Genesis** "take place at time zero". **Genesis** should be called *exactly once* at the beginning of each experiment to start the simulation clock. **Genesis** requires no argument: **(Genesis)**

The **Terminate** construct is called *exactly once* at the end of each experiment. It indicates that simulation is complete. **Terminate** requires no argument: **(Terminate)**

In addition to all the statements described above, BDL permits the user to insert C code (surrounded by "{" and "}") anywhere a statement is permitted. This allows the user to perform tasks that are not easily specified using BDL.

4.3. Building Abstract Software Models

The development of abstract software models benefit at least three important research activities in parallel processing:

• *PERFORMANCE EVALUATION OF ALTERNATIVE PROBLEM FORMULATION STRATEGY* —

Usually, there is more than one way in which an application can be formulated as computing processes to be executed in parallel. The ability to model these alternatives quickly and evaluate their performance on target parallel architectures enables software bottlenecks to be located long before tens of man-years have been spent on actual implementation.

- *PERFORMANCE PREDICTION ON NOVEL HARDWARE SYSTEM ARCHITECTURE* — In the design of large special-purpose hardware systems (such as the Data Management System of Space Station Freedom), the choice of processors and communication links has to be such that they can support the software that is going to be run on it. Unfortunately, these decisions have to be made before the software has actually been written; only abstract functional specifications exist. The ability to build approximate models from software specification and observe their performance impact on different processors and communication links will help determine suitable candidates.
- *DEVELOPMENT OF LOAD-BALANCING ALGORITHMS* — Finally, with the development of resource management algorithms for highly parallel systems, candidate strategies have to be tested over a wide range of hardware architectures and applications with different behavior characteristics to determine its performance and robustness. Simulating program execution at the instruction level is unnecessary and too time consuming. BDL models preserve program behavior, and at the same time, enable simulation to proceed quickly, allowing many experiments to be performed rapidly.

Building parallel program models with BDL involves four basic steps:

i. study and understand the program text and structure;

ii. specify the program model in BDL;

iii. identify the portion of computation that can be abstracted:
- use the **Run** statement for data-independent computations; or
- use probabilistic branches otherwise;

iv. simulate the model and compare its behavior with the actual program:
- find out the limitations of the model; and
- make modifications if necessary.

This process is best explained by using two examples — the *N-body problem* and *Quick-sort.*

4.3.1 The N-body Problem

N heavy bodies are suspended in 3-dimensional space at some initial coordinates. They are subsequently released simultaneously and allowed to move under the influence of gravitational forces they exert on one another. The problem: plot their trajectories.

A possible distributed formulation involves creating one player for each **body** (as shown in Figure 4-2). The computation begins by having a *start* message sent to each player. Each player responds by sending its current position[2] to the other (N-1) players. Whenever the new position of another body is received, the internal states of the receiver are updated. Its new position and velocity are calculated, recorded locally and transmitted back to the sender.

```
(DefPlayer Body                                          PLAYER-TYPE DEFINITION
   (init start update)                                            message names
   (b1 ... bn-1)                                          mail-addresses and
   (my_coords my_velocity crd1 crd2 ... crdn-1)                numerical states
   (init                                                         INITIALIZATION
      (record b1 b2 ... bn-1                          get neighbors' addresses
        crd1 crd2 ... crdn-1 ))                           & initial positions
   (start                                                          START MOVING
      (post b1 update my_coords)                     tell everybody where I am
      ...                                            and wait for their responses
      (post bN-1 update my_coords))
   (update                                                new position arrived!
      (if (= sender b1) (record crd1)          find sender & update its position
      ...
☞     (setq my_coords                                       compute my position
        (funcall compute_new_coords self))
☞     (setq my_velocity                                        and new velocity
        (funcall compute_new_velocity self))
      (post some_plotter update my_coords)                                 plot!
      (reply update my_coords)))                          and reply the sender
```

Figure 4-2. BDL Schematic of a "Body" Player in the N-Body Problem

Two observations can be made:

a) The most time consuming statements in the program text are the ones which compute the new position and velocity (as indicated by "☞").

b) The time spent computing these two statements is <u>independent</u> of the actual position of all the other players.

If the statements in question were replaced (as shown in Figure 4-3) with NO-OP statements that *simply consume time* — `(run some_time)`, the interaction of the program with the

[2] In order to keep the code simple, we are only keeping track of one dimension of the bodies; the CRD state variables could easily be duplicated to hold the other dimensions of the bodies.

multiprocessor does not change *to an outside observer*. In other words, the number, sequence and pattern of messages delivered and the demand posed on the processor by each player remains identical to that of the actual program. As far as simulation is concerned, advancing the "clock" using the **Run** statement is much more economical than emulating the actual instructions that solve the differential equations. Therefore, simulating this program model will enable us to minimize turn-around time while preserving program behavior as much as possible.

```
(DefPlayer Body                                    PLAYER-TYPE DEFINITION
   (init start update)                                      message names
   (b1 ... bn-1)                                     same mail-addresses
   ()                              no need to store actual coordinates
   (init                                                   INITIALIZATION
      (record b1 b2 ... bn-1))                  get neighbors' addresses
   (start                                                    START MOVING
      (post b1 update)                          tell everybody where I am and
      ...                                       wait for their responses
      (post bn-1 update))
   (update                                          NEW POSITION ARRIVED!
      (run 2)                                time spent recording new position
      ...
☞     (run 40)                                time spent compute my position
☞     (run 50)                                            and new velocity
      (post some_plotter update)                          pretend to plot!
      (reply update)))                               and reply the sender
```

Figure 4-3. BDL Abstraction of a "Body" in the N-Body Problem

A few more points need to be made with regard to using the **Run** statement in conjunction with any multiprocessor model for the purposes of simulation:

1. CPU overhead incurred for message passing (**Post**, and **Record**), blocking (**Wait**) and player creation (**Make**) does not need to be accounted for using the **Run** statement. Such parameters, together with the time involved in sending a packet across sites, will be supplied by multiprocessor model (see Table 3-2 in Section 3.3).
2. User-specified computation is *free* — this includes the use of **If**, **Progn**, **Repeat** and **Branch** statements as well as any mathematical operations.
3. The amount of time actually required for a computation can be obtained either by experimentation or estimation:
   - *experimentation* — measure the actual execution time a portion of the program requires at one site of the target multiprocessor; or

- *estimation* — generate the corresponding machine instructions (by a compiler) and estimate the actual number of cycles needed based on a processor architecture.

4. The amount of time units is only meaningful when compared to the delay for message sending and other CPU overhead charges.

4.3.2 Quicksort

```
(DefPlayer quickSort                                    PLAYER-TYPE DEFINITION
  (init done)                                                    message names
  (left right master)                                            acquaintances
  (pivot                                                         variable list
    Unsorted_Array Array_Size Left_Array Right_Array)
  (init                                                         INITIALIZATION
    (record Unsorted_Array master)                                get an array
    (setq Array_Size (funcall compute_array_size)              compute its size
    (if (<= Array_Size 1)                                     check array size
      (progn                                      if the above condition is true
        (post master done Unsorted_Array)         send array back to parent
        (destroy))                              finish, therefore self destruct!
      (progn                                                         otherwise
        (setq pivot (funcall choose Unsorted_Array))
☞       (setq Left_Array    partition & exchange array members to get "left" array
          (funcall create_lft_array pivot Unsorted_Array)
☞       (setq Right_Array    partition & exchange array members → "right" array
          (funcall create_rght_array pivot Unsorted_Array)
        (make left quickSort)                       create a player to sort left array
        (make right quickSort)                     create a player to sort right array
        (post left init Left_Array self)                       send off left array
        (post right init Right_Array self)                    send off right array
        (wait done)                 wait for acknowledgement from one of the players
        (record Left_Array)                       left or right array is not important
        (wait done)                   wait for acknowledgement from the other player
        (record Right_Array)                      left or right array is not important
☞       (setq Unsorted_Array                              merge the two sub-arrays
        (funcall merge_arrays pivot Left_Array Right_Array))
        (post master done Unsorted_Array)         return sorted array to parent
      (destroy)))))                                                         exit
```

Figure 4-4. BDL Specification of Quick-Sort

An array of numbers is to be sorted. An element of the array (say α) is chosen to partition the array into two sub-arrays, with elements in one sub-array ≤ α and the other sub-array elements > α. This procedure is repeated recursively until the size of the sub-array is less than or equal to 1. The sub-arrays are then reassembled recursively into a single sorted array.

In a distributed environment, one player is created for each Quicksort process (as shown in Figure 4-4). The computation begins by checking whether the size of an array is less than or equal to 1. If the size is greater than 1, the partition and exchange procedures will be processed, and a player will be created to sort each sub-array. The calling function will wait until both of the called functions finish their processes, then merges the two sub-arrays into a single array.

```
(DefPlayer quickSort                                         PLAYER-TYPE DEFINITION
  (init ack)                                                          message names
  (left right master)                                                 acquaintances
  (Array_Size                                        array size and estimated time
    Partition_Time Merge_Time)             per element to partition & merge arrays
  (init                                                              INITIALIZATION
    (record Array_Size Partition_Time Merge_Time master)
    (if (<= Array_Size 1)                                          check array size
      (progn                                            if the above condition is true
        (post master ack)   send the acknowledgement back to the calling function
        (destroy))                                                             exit
      (progn                                                if the condition is false
☞       (run (* Partition_Time Array_Size))        time spent partitioning array
        (make left quickSort)         create a "left" player to sort "left" sub-array
        (make right quickSort)      create a "right" player to sort "right" sub-array
        (post left init (/ Array_Size 2)           send parameters to left player
              Partition_Time Merge_Time self)
        (post right init (/ Array_Size 2)         send parameters to right player
              Partition_Time Merge_Time self)
        (wait ack)                      wait for acknowledgement from one of the players
        (wait ack)                        wait for acknowledgement from the other player
☞       (run (* Merge_Time Array_Size))             time spent merging two arrays
        (post master ack)                send acknowledgement back to the caller player
        (destroy)))))                                                          exit
```

Figure 4-5. BDL Abstraction of Quick-Sort

Two observations similar to the *N-body problem* can be made:

a) The most time consuming lines in the program text are the ones which partition and merge arrays. (as indicated by "☞").

b) The time spent on these statements is <u>independent</u> of the partition and merge processes of all the other players.

As far as simulation is concerned, sorting a real array is not required. The time needed to partition and merge arrays can be estimated as proportional to the array size. If the lines in question were replaced (as shown in Figure 4-5) with a **Run** statement, program behavior *as observed by the multiprocessor* does not change.

4.4. Summary

BDL can be used to model parallel computations which exploit parallelism explicitly at the *process/procedure* level. There are no shared data-structures. Players communicate via asynchronous message passing. The following computing paradigms can be modelled easily using players [12]:

- PARALLEL CODE-BODIES — e.g. fork/join [13, 14], parBegin/parEnd [15];
- COMMUNICATING PROCESSES — e.g. Concurrent Pascal [16] and CSP [17];
- MULTILISP [18], *futures* [10] and *streams* [19];
- REMOTE PROCEDURES [11]; and
- a subset of the ACTOR programming paradigm [20] known as the *serializers* [21].

BDL also provides the necessary constructs to model the following aspects of parallel programs:

- behavior of individual players — models the different functionality of various computing agents in the parallel computation;
- player creation — both compile-time declaration and run-time creation;
- communication — for data transfer and synchronization between computing agents; and
- processing in response to messages received — in order to construct an accurate model, the processing time can be gathered by monitoring actual program execution or by counting machine instructions.

As far as building program models is concerned, the data-dependent behavior of programs remains the biggest challenge for model builders. Although the use of probabilistic branches may solve some simple cases, it is, in general, very difficult to decide what an *appropriate level* of modelling should be.

For example, two alternative player representations of a "file" are shown in Figure 4-6. The model on the left checks whether the file has already been reserved before granting a user write-access. The one on the right, however, is built based on an observation that "only (say) 5% of users try to gain access to files already reserved". BDL allow models of either level to be specified. It is up to the researcher to decide, recognize and remember what level of abstraction his/her model was constructed for. A model should never be interpreted beyond its limitations.

```
(DefPlayer File
 (... reserve ...)
 (my_writer ...)
  ...
 (reserve
  (if (<> my_writer NIL)
   (reply ack NIL)
   (progn (setq my_writer sender)
   (reply ack T))))
 ...)
```

```
(DefPlayer File
 (... reserve ...) msgs it understands
 (my_writer ...)   acquaintance names
  ...
 (reserve              RESERVE msg received
  (branch
   (.05 (reply ack NIL))       REFUSE!

   (.95 (reply ack T))))       ACCEPT!
...)
```

Figure 4-6. Two BDL Models of a "File" Player

5. Conclusions and Future Research

AXE, *AN EXPERIMENTATION ENVIRONMENT,* is created to facilitate research with resource management strategies for parallel systems. It provides an integrated environment for the following activities:

- COMPUTATION MODEL SPECIFICATION using BDL — a Behavior Description Language for parallel computations. BDL can be used to describe computation based on various programming paradigms such as CSP, remote procedures, data-flow, and actors.
- MULTIPROCESSOR ARCHITECTURE SPECIFICATION — By changing certain simple parameters, the architecture of the hardware may be modified. This includes inter-connection topology, network speed, routing algorithms as well as operating system scheduling algorithms.
- SIMULATION — A discrete-time event-driven simulator based on CSIM is responsible for predicting the execution time of BDL program models on various multiprocessor models.

- DATA COLLECTION — Data that indicate program behavior and resource utilization/contention are gathered automatically. These data may be used for evaluation of resource management strategies as well as software and hardware architecture alternatives.
- EXPERIMENTATION — The researcher is able to study resource management strategies as well as various issues in parallel processing such as problem formulation, alternate hardware architectures, and operating system algorithms.

During simulation, the activities on the multiprocessor are displayed dynamically via a color monitor. These include message transmission, length of "ready-queue" at each processing element as well as overall system load.

The *AXE Modelling Package* is being augmented to model Intel Hypercube δ (iPSC/3) and various multiprocessor testbed configurations for the NASA High Performance Computing and Communications Program. Various panels from the *AXE Visualization Package* will be integrated with other tools for displaying data collected from these multiprocessor testbeds.

Acknowledgements

The authors would like to acknowledge Roger Bartlett of Digital Equipment Corporation, who wrote most of the *AXE Visualization Package*, Christopher Jones of Stanford University who developed the token-ring multiprocessor model, Arthur Ieumwananonthachai and Lon. C. Chu of University of Illinois Urbana-Champaign who developed the iPSC/2 routing model, Sonie Lau of NASA Ames Research Center who wrote an earlier version of the *Visualization Package* and Rafael Villegas of NASA Ames Research Center who wrote an earlier version of the BDL compiler.

References

[1] J. C. Yan and S. F. Lundstrom. "The Post-Game Analysis Framework — Developing Resource Management Strategies for Concurrent Systems". IEEE Transactions on Knowledge and Data Engineering, pages 293 - 309, vol.1, no.3, September 1989.

[2] H. Schwetman. "CSIM: A C-Based, Process-Oriented Simulation Language". In *Proceedings of the Winter Simulation Conference '86,* Washington DC, 1986.

[3] H. Schwetman. "CSIM Reference Manual (Revision 14)". MCC Technical Report ACT-ST-252-87, Rev. 14. Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin TX 78759. March 1990.

[4] A. Malony and D. Reed. "Visualizing Parallel Computer System Performance". Report UIUCDCS-R-88-1465, Department of Computer Science, University of Illinois at Urbana-Champaign, September 1988.

[5] B. Delagi, N. Saraiya, S. Nishimura, and G. Byrd. "Instrumented Architectural Simulation". Report KSL-87-65, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, November 1987.

[6] C. E. Fineman and J. C. Yan. "The Axe User Manual". NASA Contrator Report. *In preparation.*

[7] H. D. Brown, E. Schoen, and B. A. Delagi. "An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures". Report KSL-86-69, Knowledge Systems Laboratory, Department of Computer Science Stanford University, Oct. 1986.

[8] W. J. Dally. "Wire-Efficient VLSI Multiprocessor Communication Networks". in Advanced Research in VLSI, In *Advanced research in VLSI: Proceedings of the 1987 Stanford Conference,* Editor, Paul Losleben, pages 390-415, 1987.

[9] P. Kermani and L. Kleinrock. "Virtual Cut-Through: A New Computer Communication Switching Technique". Computer Networks, 3:267-286, 1979.

[10] H. G. Baker, Jr. and C. Hewitt, "The Incremental Garbage Collection of Processes". Massachusetts Institute of Technology, AI Lab., AI Working Paper 149, July, 1977.

[11] B. J. Nelson, "Remote Procedure Call", Xerox Palo Alto Research Center, Palo Alto, California, Tech. Report CSL-82-8

[12] J. C. Yan. "Parallel Program Behavior Specification and Abstraction using BDL". CSL-TR-86-298, Computer System Laboratory, Stanford University, August 1986.

[13] M. Conway, "A Multiprocessing System Design", *Proceedings of the AFIPS Fall Joint Computer Conference*, 1963.

[14] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM,* March 1966.

[15] E. W. Dijkstra, "*Cooperating Sequential Processes*", Tech. Report EWD-123 Technological Universty, Eindhoven, The Nedelands.

[16] P. B. Hansen, "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering,* June 1975.

[17] C. A. R. Hoare. "Communicating Sequential Processes". *Communications of the ACM,* August, 1978.

[18] R. Halstead. "Parallel Symbolic Computing". *Computer Magazine,* 19(8):35-43, August, 1986.

[19] K-S. Weng, "Stream-Oriented Computation in Data Flow Schemas". TM 68, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1975.

[20] G. A. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems". TR 844, PhD Thesis, Massachusetts Institute of Technology, AI Lab., 1985.

[21] R. Atkinson and C. Hewitt. "Specification and Proof Techniques for Serializers". Memo 438, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1977.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-177582	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Modelling Parallel Programs and Multiprocessor Architectures with AXE		5. Report Date May 1991
		6. Performing Organization Code
7. Author(s) Jerry C. Yan and Charles E. Fineman		8. Performing Organization Report No. A-91135
		10. Work Unit No. 505-64-54
9. Performing Organization Name and Address Sterling Federal Systems, Inc. 1121 San Antonio Road Palo Alto, CA 94303		11. Contract or Grant No. NAS2-13210
12. Sponsoring Agency Name and Address Ames Research Center Moffett Field, CA 94035-1000		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes
Point of Contact: Bob Carlson, Ames Research Center, MS 233-15, Moffett Field, CA 94035-1000
(415) 604-6036 or FTS 464-6036

16. Abstract

AXE, An Experimentation Environment for Parallel Systems, was designed to model and simulate for parallel systems at the "process level." It provides an integrated environment for specifying computation models, multiprocessor architectures, data collection and performance visualization. AXE is being used at NASA Ames Research Center for developing resource management strategies, parallel problem formulation, multiprocessor architectures, and operating system issues related to the High Performance Computing and Communications Program. AXE's simple, structured user-interface enables the user to model parallel programs and machines precisely and efficiently. Its quick turn-around time keeps the user interested and productive.

AXE models multicomputers. The user may easily modify various architectural parameters including the number of sites, connection topologies, and overhead for operating system activities. Parallel computations in AXE are represented as collections of autonomous computing objects known as players. Players may be used to represent parallel code-bodies, communicating sequential processes, Halstead's futures, remote procedures, and a subset of Hewitt's actors. A Behavior Description Language (BDL) is used to specify player programs.

Performance data of the multiprocessor model can be observed on a color screen. These include CPU and message routing bottlenecks, and the dynamic status of the software.

17. Key Words (Suggested by Author(s)) Simulation Performance evaluation Parallel processing High performance computing		18. Distribution Statement Unclassified-Unlimited Subject Category – 62	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 42	22. Price A03

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161