DPOS Programming Manual

John D. Evans

UUCS-90-020

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

October 9, 1990

DPOS Programming Manual

John D. Evans

Department of Computer Science
University of Utah
Salt Lake City, Utah

1 Introduction

This manual describes the basic concepts of the DPOS Metalanguage and the programming language DPOS Scheme.

2 DPOS Concepts

DPOS is a metalanguage for defining parallel program networks based on the common requirements of distributed parallel computing that is portable across languages, modular, and highly flexible. The system uses the concept of **stratification** to separate process network creation and the control of parallelism from computational work. Individual processes are defined within the **process object** layer as traditional single threaded programs without parallel language constructs. Process networks and communication are defined graphically within the **system** layer at a high level of abstraction as recursive graphs. Communication is facilitated in DPOS by extending message-passing semantics in several ways to implement highly flexible message-passing constructs. DPOS processes exchange messages through bi-directional **channel** objects using guarded, buffered, synchronous and asynchronous communication semantics.

A DPOS program is defined as a network of active processes called Process Objects (POs) and communication lines called Channels that are grouped into subnetworks called Network Modules (NMs). In DPOS a communicating process network model is used.

2.1 Process Objects

Process Objects (PO) are single threaded program functions with calling parameters identical to traditional sequential program functions. Multiple Process Objects execute concurrently. These active objects employ much of the modularity, encapsulation of function, and encapsulation of data found in sequential object-oriented programming.

Process Objects communicate with each other by sending and receiveing messages. DPOS message semantics are similar to file I/O in sequential programming.

Process Objects may **block** while attempting to send to or receive messages from channels if the channel transaction cannot be completed immediately. This causes the execution to suspend, but leaves the runtime stack intact. Progress may resume at a later time if the transaction can be completed. This blocking activity is transparent to the Process Object program.

The connection links (Channels) between Process Objects appear as variables passed in as calling parameters similar to files or streams in traditional programming. Process Objects communicate with each other via channel accessor functions like **send** or **receive**. No additional syntax or language extensions are required, since simple function call syntax is used. The control flow of Process Objects is internal. The progress of computation, however, may be controlled by regulating message traffic into and out of the Process Object causing the PO to block waiting for communication to proceed. The synchronization required for communication is controlled by the communication channel. Because of this a PO may follow a bounded sequential computation or may be an unbounded cyclical computation (like an operating system process) that is I/O driven via its communication channels.

The creation of Process Objects is specified within the parent Network Module. The termination of process objects occurs when the execution of the code segment for the process object terminates.

The sequential nature of Process Objects allows them to be developed and debugged individually as separate programs before integration into a network module. Simple terminal input and output is substituted for channel communication during sequential debugging. A library of channel definitions in the

base language has been developed for sequential debugging (not discussed here).

Global data structures are not defined within the DPOS model. The use of channel objects to implement data that is shared between many processes is discussed below (see subsections 2.2 and 8.1).

2.2 Channels

Communication and synchronization between Process Objects is accomplished by message-passing. DPOS processes exchange messages through bi-directional **channel** objects.

In DPOS a channel is treated as a separate object in the object-oriented sense and not just as a communication relationship (as it is in most channel oriented communication systems). This allows the semantics of channels to be extended in several ways. The semantics of channel communication is an attribute of the channel. DPOS channels encapsulate both functional semantics and in some cases data storage.

This encapsulation has several advantages:

1. Communication is indirect. Allowing channels to be accessed by multiple sender and receiver processes. The arbitration for access is handled within the channel object. For example, the merging and splitting streams of data is trivially implemented using shared channels. Multiple process access to channels allows channels to be used to implement shared data values.

2. Indirect communication also facilitates dynamic process creation. It allows new processes to be added to a network that communicate through existing channels without notifying existing processes.

3. It removes the need for language constructs to implement communication semantics. Because of this, the PO definitions of a DPOS program need no extensions beyond traditional single threaded programming constructs.

4. It allows bi-directional communication.

5. It allows multiple classes of channels. In DPOS the class of a channel specifies the semantics of communication for that particular channel. DPOS channel class include: synchronized guarded input, synchronized guarded output, asynchronous, buffered, and synchronous.

2.3 Network Modules

Network Modules (NMs) are abstractions used for defining subnetworks in DPOS programs. Network Modules are composed of Process Objects, Channels and other nested Network Modules. Network Module classes are defined as graph structures (see Figure 1). Network Module definitions have local environment and have formal parameters that correspond to data values and channel instances. Invoking an NM requires actual parameter arguments to be provided. The arguments are the actual access channels that connect to the NM as well as any required values computed within the scope of the invoking environment. Network Modules may be nested and recursively or mutually recursively nested. In the traditional object-oriented framework, a Network Module definition constitutes a class definition and may be instanced numerous times in the definition of a process network.

Global values are not defined within DPOS. The only exterior environment visible from within a Network Module consists of the actual arguments passed in at the instantiation of the NM. A Network Module then is completely encapsulated by the actual arguments and peripheral channels and may be analyzed as a unit. This modularity allows a Network Module to be developed and debugged as a unit by instantiating it and its peripheral channels and without creating the outlying program network.

A Network Module instance might not be instantiated when the parent NM is instantiated. The instantiation of the NM may be delayed until a demand is made for its creation. Delayed Network Modules are instantiated whenever data flow occurs in one of its peripheral channels. Alternatively the instantiation may be conditional in which case a constraint condition is evaluated within the parent NM environment to determine whether or not the instance is to be instantiated. Constraint conditions and instantiation delays are specified within the parent NM definition. These properties allow process networks to be specified in a manner similar to abstract data types such as trees and graphs in high-level languages. For example, a generic binary tree Network Module may be defined which is then used to define a specific, irregular extended tree process network.

3 Example: Producer and Consumer

This section describes a short example DPOS program using DPOS Scheme and DPOS program graphs. DPOS Scheme is a concurrent Scheme dialect and is described in section 4.

The example program presents the producer/consumer problem. A **producer** process

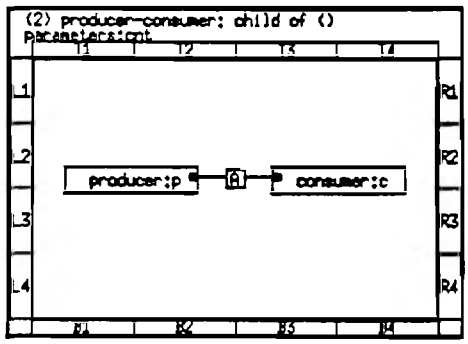

Figure 1: producer-consumer Network Module

```
(define (producer-po CHAN cnt)
  (do ((a 0 (+ a 1)))
      ((= a cnt))
    (display (list 'produced a))
    (newline)
    (send CHAN a))
  (display 'producer-quits))

(define (consumer-po CHAN cnt)
  (let ((input #f))
    (do ((a 0 (+ a 1)))
        ((= a cnt))
      (set! input (receive CHAN))
      (display
       (list 'consumer 'receives a input))
      (newline)))
  (display 'consumer-quits))
```

Figure 2: Process Object Code

```
load channel.scm
DEFS
load producer-consumer.dp
load producer.dp
load consumer.dp
BODIES
(producer-consumer 3)
ENDBODIES
```

Figure 3: Producer-consumer Run File

```
(produced 0.000000)
(produced 1.000000)
(consumer receives 0.000000 0.000000)
(produced 2.000000)
(consumer receives 1.000000 1.000000)
producer-quits
(consumer receives 2.000000 2.000000)
consumer-quits
```

Figure 4: Program Output

creates data objects and sends them to a **consumer** process. The Network Module for this program is shown in Figure 1. This Network Module contains three objects. Two processes are created **producer** and **consumer**. The producer PO is of class **producer** and has instance name **p**. The consumer PO is of class **consumer** and has instance name **c**. A channel is also created (labeled A). The **producer-consumer** Network Module has a single parameter **cnt** which is passed to each of the Process Objects. The source code for the individual processes is shown in Figure 2. The producer **sends** to the channel and the consumer **receives** from the channel. A run file for the program is given in Figure 3 and the output generated is shown in Figure 4.

4 Basic DPOS Scheme Concepts

DPOS Scheme includes a subset of standard sequential Scheme features, and introduces several additional features. DPOS Scheme is a concurrent Scheme dialect. It includes features for generating and executing parallel Scheme. The parallel features specify process creation, process management and message passing and synchronization constructs.

The language was designed to be programmed using the DPOS metalanguage and graphical interface to generate parallel process networks. This section describes the basic concepts of DPOS Scheme. An overview and list of DPOS Scheme standard features is presented in section 10.

4.1 Top Level Programming

There is no top level read-eval-print loop in DPOS Scheme. In DPOS Scheme a single run file (see Figure 3) is evaluated which specifies all top level definitions. Definitions may be specified directly or by loading Process Object and Network Module files generated

by the DPOS graphical interface. A sequence of command forms follows the definitions. The command forms are evaluated in the environment created by the top level definitions.

The command forms specify standard sequential Scheme computations and also the creation, management and communications of concurrently executing Scheme processes. DPOS Scheme allows both shared memory and distributed memory parallel programming. In many programs the distinction between shared and distributed memory semantics does not have any effect on the DPOS program definition.

DPOS Scheme may be used to implement DPOS Network Module and Process Object definitions. DPOS Network Module and Process Object definitions are generated by the DPOS Graphical Interface as DPOS Scheme load files. The contents of Network Module definition files are not intended to be modified by programmers and are not discussed here.

4.2 Process Object Definitions

The definitions of DPOS Process Objects are specified by programmers using a text editor. Process Object definitions are sequential Scheme functions (see Figure 2). Each Process Object has access to all the function definitions in the top level environment. The only contact between multiple Process Objects or between Process Objects and the outer Network Module Environment is the actual argument list supplied to the function at evaluation. The argument list contains channels and data parameter values. DPOS Scheme Process Objects comply with the definition of Process Objects give in subsection 2.1. The Process Object definitions may use any of the features listed in subsection 10.2. In addition, Process Object definitions may use concurrent extensions listed in subsection 10.3.

The Process Object Source files are read by the DPOS Graphical interface, modified and output as Process Object load files. Process Object load files have the extension **.dp**. Instances of Process Objects are specified within Network Module definitions. The arguments supplied to PO instances are also specified with the parent Network Module definition.

4.3 Network Module Definitions

The definitions of DPOS Network Modules are specified by programmers using the DPOS Graphical Interface (see Figure 1). The source code for Network Module definitions is generated by the graphical interface in Network Module load files. Network Module load files have the extension **.dp**. Network Module load files are not intended to be modified by programmers.

DPOS Scheme Network Modules comply with the definition of Network Modules given in subsection 2.3.

Network Module definitions have internal PO, NM and channel instances specified by the programmer. All instances are named. Names are specified by the programmer. Network Module definitions have parameters specified by the programmer. Parameters for the nested instances are specified by the programmer and may contain the results of expressions using local values and channel instances.

4.4 Channel Definitions

Channel instances are specified by programmers using the DPOS Graphical Interface as part of a network module definition. Three channel classes are presently supported by DPOS Scheme: **a_channel**, **b_channel**, and **i_channel**. DPOS Scheme channels comply with the definition of channels given in subsection 2.2. Channel communication is specified in the Process Object definition files using the syntax and semantics listed below.

A_channel class channels implement asynchronous communication. A single item may be buffered in the channel. The operations on **a_channels** are:

1. (**send** chan data): The send operation inserts data into chan if chan is empty. If chan is not empty then the PO blocks until it becomes empty.

2. (**receive** chan): Returns the contents of chan and clears the channel. If chan is empty then the PO blocks until it becomes full.

3. (**chan-read** chan): Returns the contents of chan but does not clear the channel. If chan is empty then the PO blocks until it becomes full.

Subsection 8.1 demonstrates all **a_channel** operations.

B_channel class channels implement asynchronous buffered communication. A buffer size limit is specified when the channel is specified in an NM definition. The operations on **b_channels** are:

1. (**send** chan data): The **send** operation inserts data into chan if chan contains less than **size** elements. If chan is full then the PO blocks until it contains less than **size** elements.

2. (**receive** chan): Returns the first item in chan (FIFO order) and removes it from the channel.

If chan is empty then the PO blocks until it becomes non-empty.

Subsections 8.1 and 8.3 demonstrate **b_channel** operations.

I_channel class channels implement guarded input communication. Communication is synchronized and no messages are buffered. A **size** value is specified when the channel instance is defined within an NM definition. **Size** specifies the number of message types recognized by the channel. Message types are simple integers. If **size** is 3 then the channel will recognize message types 0, 1 and 2. The operations on **i_channelss** are:

1. (**read-g** channel type-list): The type-list is a list of integers. The integers must be between 0 and **size**. These integers specify the types of messages that are acceptable. Returns a message of a specified type from the channel. The return value is a list (**type data**), where type is one of the types in type-list.

2. (**write-g** channel type data): Sends a message of "type" to "channel" which contains "data".

If no writer of an acceptable type is blocked trying to **write-g** to the channel then the reader blocks until one comes along. If a reader is not blocked trying to **read-g** the message type of a writer then writer blocks. Readers are serviced by the channel on a first come first served basis. This means that if reader1 is blocked trying to read a type 1 message then no other readers may be serviced until a type 1 writer comes along. Subsections 5.2 and 8.2 demonstrate **i_channel** usage.

5 Example Programs

This section presents example programs using DPOS Network Modules and DPOS Scheme Process Object definitions. Run files for the DPOS Scheme simulator are also included. The programs are not meant to display highly efficient parallel programs, but only to demonstrate basic DPOS parallel programming concepts.

5.1 Multiple Producers and Consumers

This example is a simple extension of the producer/consumer problem to handle multiple producers and consumers. It demonstrates the use of nested Network Module definitions. It also demonstrates the

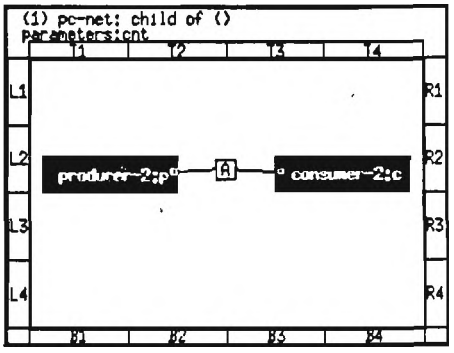

Figure 5: Pc-net Network Module

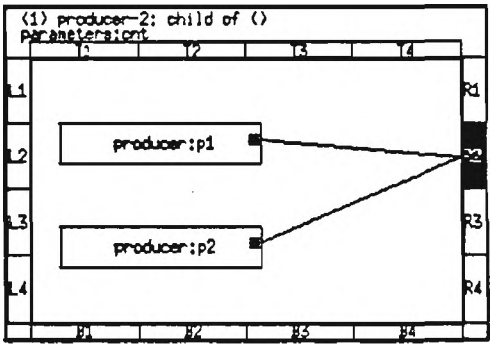

Figure 6: Producer-2 Network Module

```
(produced 0.000000)
(produced 0.000000)
(produced 1.000000)
(consumer receives 0.000000 0.000000)
producer-quits
(consumer receives 0.000000 1.000000)
(produced 1.000000)
(consumer receives 1.000000 0.000000)
consumer-quits
producer-quits
(consumer receives 1.000000 1.000000)
consumer-quits
```

Figure 7: Program Output

use of multiple senders and receivers accessing a channel.

The Network Modules for this program are shown in Figures 5 and 6. The producer class POs in the **producer-2** NM (Figure 6) are connected to the output port R2. The port R2 corresponds to the port shown on **producer-2:p** in Figure 5. When the program is run the producer POs have access via this port to the channel in the outlying **pc-net** NM. The value of "cnt" in **pc-net** is passed as an argument to the nested network modules and in turn to the nested Process Objects. The **consumer-2** Network Module is similar to the **producer-2** Network Module shown in Figure 6.

When the program is run using the command (**pc-net 2**), a **pc-net** NM is created. This causes the creation of the channel and the nested NMs which in turn create the nested POs. All POs access the channel. The synchronization of accesses is controlled by the channel and allows only a single process at a time to operate on the channel. This results in serialization if simultaneous accesses occur. It also results in a nondeterministic merging of the output streams of the producers into the channel, and the nondeterministic branching of input streams into the consumers. The program output is shown in Figure 7. The sequence of output statements is not deterministic. Running the program several times could result in slight variations in the output sequence. The channel in this and the previous examples is an asynchronous channel. This channel class allows a sender to deposit a message and continue without waiting for a receiver process. This kind of message passing allows a higher degree of parallelism than strictly synchronized message passing.

5.2 Bank Account Program

This example program presents a bank account management problem. In this problem a bank account process receives requests for deposits and withdrawls. Deposits are always possible and are serviced in FIFO order. Withdrawls are also serviced in FIFO order, however, if insufficient funds are available a withdrawl will be delayed until funds become available.

The Network Module for this program is shown in Figure 8 and the source code for the process objects is shown in Figure 9. In Figure 8 the parameters "d-cnt" and "w-cnt" indicate how many deposits and withdrawls each process is to attempt. Figure 10 shows program output for the invocation of (**bank-net 2 1**).

The solution illustrates the use of a guarded input channel. The solution becomes much more complex to implement when other channel classes are used. The channel recognizes three message types: **deposit**, **w-request**, and **w-ok**. **Depositor** and **withdrawer** processes always **write-g** to the channel and the **bank** always **read-gs** to the channel. A **depositor** process uses **deposit** message type. A **withdrawer** process uses **w-request** message to enter a request, then uses **w-ok** message to acknowledge the transaction.

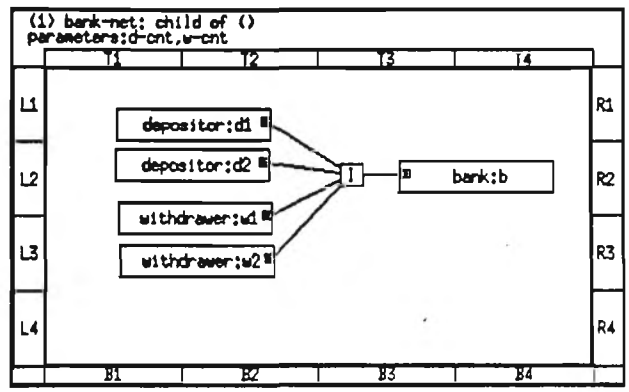

Figure 8: Bank-net Network Module

The **bank** process manages a list of allowable message types that it will accept. Initially the **bank** will accept **w-request** and **deposit**. If a **w-request** is received it sets the list to include only **deposit**. When sufficient funds to service a withdrawl are available, **w-ok** is added to the guard list. When the **w-ok** is received from the **withdrawer** the list is reset to its original form.

Notice that receipt of funds by the **withdrawer** is signified by the successful receipt of a message by the **bank**. This kind of transaction is common in guarded message sending, but takes some getting used to.

5.3 Factorial Program

This example program computes **factorial** of N (see Figures 11, 12, 13, and 14). It presents no new DPOS concepts, but demonstrates a typical parallel programming concept. It uses a controller process (**fcontrol:fc**) and a group of servant processes (**class f-unit**) that carry out work tasks defined by the controller. A servant processes receives limits for a subset of the factorial computation from channel **TOP-CH**. When each has completed its subset, it receives the total from the next left servant from channel **LEFT-CH**, multiplies in its subtotal and sends the result to **RIGHT-CH**. The controller receives the total from the third servant "F3".

```
(define (bank-po BANK-CH)
 (let
  ((balance 0)
   (pending #f)
   (guards (list w-request deposit)))
  (display (list 'start-balance balance))
  (do ((request (read-g BANK-CH guards)
                (read-g BANK-CH guards)))
      (#f)
   (cond
    ((= (car request) w-ok)
     (set! guards (list w-request deposit)))
    ((= (car request) w-request)
     (set! pending (cadr request))
     (set! guards (list deposit)))
    (#t (set! balance (+ balance
                         (cadr request)))
        (display (list 'balance balance))))
   (if (and pending (<= pending balance))
     (begin
      (set! guards (list w-ok deposit))
      (set! balance (- balance pending))
      (display (list 'balance balance))
      (set! pending #f))))))

(define (withdrawer-po BANK-CH cnt)
  (do ((a 0 (+ a 1)))
      ((= a cnt ))
    (let ((amount (random)))
      (write-g BANK-CH w-request amount)
      (write-g BANK-CH w-ok)
      (display (list 'withdrew a amount)))))

(define (depositor-po BANK-CH cnt)
  (do ((a 0 (+ a 1)))
      ((= a cnt))
    (let ((amount (random)))
      (write-g BANK-CH deposit amount)
      (display (list 'deposit a amount)))))
```

Figure 9: Process Object definitions

```
(start-balance 0.000000)
(balance 11.900600)
(deposit 0.000000 11.900600)
(balance 15.767471)
(deposit 1.000000 3.866871)
(balance 53.135357)
(balance 35.591751)
(deposit 0.000000 37.367886)
(withdrew 0.000000 17.543608)
(balance 27.355316)
(balance 29.170748)
(deposit 1.000000 1.815432)
(withdrew 0.000000 8.236435)
```

Figure 10: Program Output

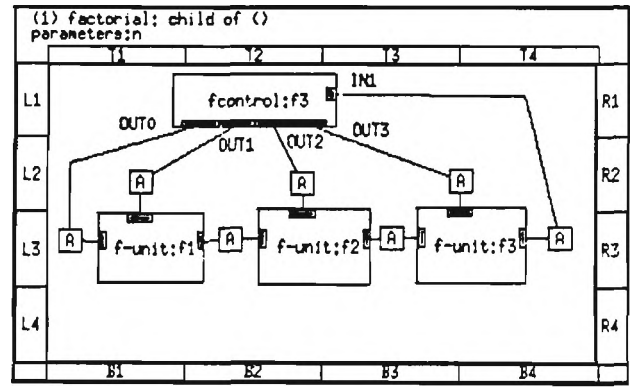

Figure 11: Factorial Network Module

```
(define (fcontrol n OUT0 OUT1 OUT2 OUT3 IN1)
  (let ((size (truncate (/ n 3))))
    (send OUT1 (list 0 size))
    (send OUT2 (list size (* 2 size)))
    (send OUT3 (list (* size 2) (+ n 1))))
  (send OUT0 1)
  (display
    (list 'factorial-of n (receive IN1))))

(define (f-unit-po LEFT TOP RIGHT)
  (let ((limits (receive TOP))
        (result 1))
    (do ((a (car limits) (+ a 1)))
        ((= a (cadr limits)))
      (set! result (* result a)))
    (send RIGHT (* (receive LEFT) result))))
```

Figure 12: Process Object Definitions

```
load channel_debug.scm
DEFS
load factorial.dp
load fcontrol.dp
load f-unit.dp
(define (factor1 n)
  (if (> n 1 )
      (* n (factor1 (- n 1)))
      1))
BODIES
(init-debug 'factorial)
(factorial 20)
(display (list 'factor1 (factor1 20)))
ENDBODIES
```

Figure 13: Factorial Run File

```
(factor1   2432902023163674500.000000)
(factorial-of
 20.000000 2432902023163674500.000000)
```

Figure 14: Program Output

6 DPOS Graphical Interface

This section presents an overview of the DPOS Graphical Interface and its use for generating simple DPOS programs. This section covers in detail the features necessary to generate the example programs already presented. The DPOS graphical interface is used to generate Network Module definitions, to define Process Object interfaces, to generate load files for Process Objects and Network Modules in target languages, and to debug programs. The discussion in this section is confined to definition of NMs, definition of POs, and the generation of load files in DPOS Scheme.

6.1 Starting the Graphical Interface

The interface uses the X-11 Window system. X-11 must be running to run the DPOS Graphical Interface. A black and white version of the interface can be run by typing:

```
dpos -bw
```

The interface also uses colors:

```
dpos -b0 lightgray -f0 black
```

Specifies a gray background and black foreground.

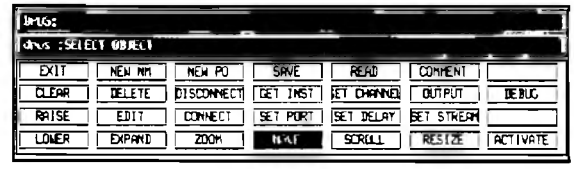

Figure 15: Top Level Menu

```
dpos -brdr lightgray -b0 lightgray -f0 black
     -b4 MediumVioletRed -f4 Navy
     -b5 MediumGoldenrod -f5 white
```

Sets up colors for debugging. More sophisticated color selection is possible, but is not covered here.

6.2 Top Level Menu

When the program is run the top level menu appears (see Figure 15). Interaction with the interface is by mouse buttons and by keyboard. Only the basic features will be discussed here. Advanced features are discussed in a later section. Top level menu commands include:

1. EXIT: Exit the interface (and don't bother saving anything).
2. NEW NM: Open a Network Module template for editing.
3. NEW PO: Open a Process Object template for editing.
4. SAVE: Save a PO or NM template definition.
5. READ: Read an existing NM or PO template definition.
6. COMMENT: Generate a comment line for an NM template.
7. CLEAR: Clear a template window for reuse.
8. DELETE: Delete a parameter menu, PO, NM, or comment instance from an NM template.
9. DISCONNECT: Disconnect an instance port from a channel or template port.
10. GET INST: Read a PO or NM instance definition for insertion into an NM template.
11. GET CHANNEL: Read an instance description for insertion into a NM template.

12. OUTPUT: Generate source code load file for an NM or PO template.

13. DEBUG: Enter the program animator for graphical debugging.

14. RAISE: Raise an instance within an NM template.

15. EDIT: Open a text menu editor for template or instance parameters.

16. CONNECT: Create a connection from an instance port to a channel instance or template port.

17. SET PORT: Toggle PO template ports.

18. SET DELAY: Toggle instance ports as delay con nections.

19. SET STREAM: Toggle channel instances a streams of channels or single channels.

20. LOWER: Lower an instance within an NM tem plate.

21. EXPAND: Expand the distances between objects within an NM template.

22. ZOOM: Enlarge the size of objects and expand as in "EXPAND".

23. MOVE: Move an instance within an NM template.

24. SCROLL: Scroll all objects within an NM template.

25. RESIZE: Resize an instance within an NM template.

26. ACTIVATE: Open a new NM template for an instance in an NM template.

6.3 Defining Process Objects

The source code for basic Process Objects and their representation as blocks in Network Modules has been discussed in previous sections (see Section 5 and Figures 6 and 8). Process Object classes are defined within DPOS by editing Process Object templates (see Figure 16). Selecting the top level menu command "NEW PO" generates a new blank Process Object template. The graphical interface specification of a PO class defines the port connections for the graphical block, the parameters for the PO and also the interface to the user defined source code.

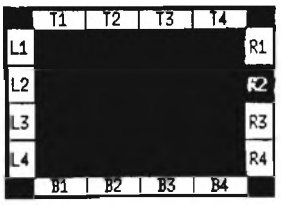

Figure 16: Process Object Template

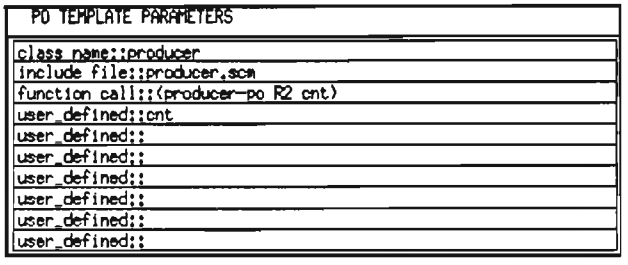

Figure 17: PO Parameter Menu

The Process Object template of Figure 16 corresponds to the **producer** PO instances in Figure 6. Selecting the "SET PORT" command sets the interace mode to toggle PO ports. Selecting a PO template port will toggle the template port on or off. In the Figure 16 port "R2" has been toggled on.

The PO parameters and source code interface are specified by editing the PO parameter menu. Selecting the "EDIT" command sets the interface mode to edit objects. Selecting the PO template will open a PO template menu (see Figure 17). The user selects one of the lines with the mouse, then uses the keyboard to specify the entry. The user must specify the PO "class name". The user may specify "user_defined" parameters. In this case the one parameter "cnt" is defined. The "include file" specifies the file which includes the user defined source code. The function call specifies a function within the source code file and the calling sequence. The "function call" in Figure 17 corresponds with the Process Object source code shown in Figure 2.

The "include file" and "function call" must be set. The name of the called function should be different from the PO class name.

Process Object templates may be saved using the "SAVE" command. "SAVE" produces a definition file which may be restored into a blank PO template using the "READ" command. Alternately, SAVED

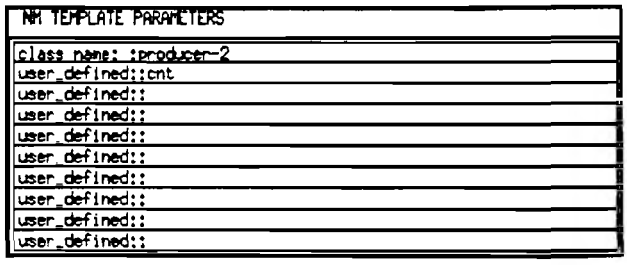

Figure 18: NM Parameter Menu

Figure 19: Channel Selection Menu

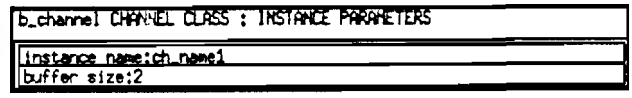

Figure 20: Buffer Channel Parameter Menu

templates may be read as instance blocks in a template (see subsection 6.6 below). Resaving a template modifies the class definition but does not affect the instances already placed within other templates. These existing instances may have to be replaced if the interface for the template is modified.

The "CLEAR" command may be used to clear out the contents of a PO template to allow another definition to be read in and edited.

6.4 Defining Network Modules

Network Module classes are defined by editing Network Module templates (see Figures 5,6, and 8). The meanings of these Network Modules has been discussed in previous sections.

The main window of a Network Module template may contain, PO blocks, NM blocks, channel blocks, comment blocks and connection lines. The graphical interface provides window management facilities for managing the blocks. The top level menu commands "ZOOM", "EXPAND", "MOVE", "SCROLL", "RAISE", "LOWER" and "RESIZE" are for manipulating blocks in template windows. The templates themselves may be resized, moved, etc. using the regular X window commands.

The contents of an NM template may be saved as an NM class definition using the "SAVE" command. "SAVE" produces a definition file which may be restored into a blank NM template using the "READ" command. Alternately, saved templates may be read as instance blocks in a template (see subsection 6.6 below). Resaving a template modifies the class definition but does not affect the instances already placed within other templates. These existing instances may have to be replaced if the interface for the template is modified.

The command "NEW NM" opens a new blank NM template for editing. The graphical interface specification of an NM completely defines the NM class. NM class parameters are specified similarly to PO class parameters by opening an NM parameter menu (see Figure 18). NM parameters include "class name" and "user_defined" parameters.

An alternate method of opening a new template for editing is to use the "ACTIVATE" command. "ACTIVATE" lets a user open a template window by selecting an NM instance that is visible within a template being edited. A new NM template is created with the class definition of the selected instance.

The "CLEAR" command may be used to clear out the contents of a PO template to allow another definition to be read in and edited.

The "COMMENT" command allows the user to enter a text string which may then be instanced within templates.

The following subsections define basic manipulations of NM templates to specify NM, PO and channel instances.

6.5 Specifying Channels

Instances of channel classes may be included in NM templates. Selecting "GET CHANNEL" opens the channel selection menu 19. Selecting a channel class puts the interface into channel instancing mode. Selecting locations in NM templates will create instances of the specified channel type in the NM template.

Channels have parameters which must be specified by the user. Channel parameter menus may be opened similarly to PO and NM parameter menus. Different channel classes have different parameter menus:

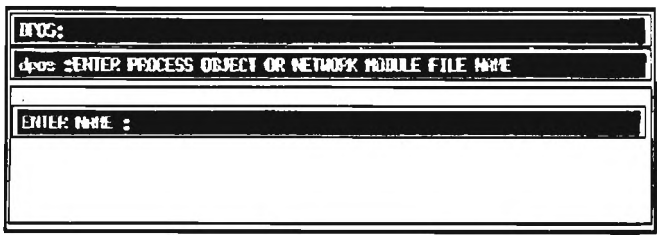

Figure 21: Get Instance Menu

Figure 22: Process Object Instance Menu

1. Asynchronous channels have only a "name" parameter.

2. Buffered channels have "name" and "buffer size" (see Figure 20).

3. Guarded input channels have "name" and "guardlist size".

6.6 Specifying Instances

NM and PO instances may be included in NM templates. Selecting "GET INSTANCE" will cause the interface to request an instance class name (see Figure 21). Once a correct PO or NM class name has been entered the user may select locations in NM templates to place instances.

Instances have parameters which must be specified by the user. Instances parameter menus are opened similarly to other parameter menus. Figure 22 shows the instance parameter menu for **producer:p2** in Figure 6. "Instance name" has been set to "p2" and the instance parameter "cnt" was a user defined parameter discussed above and specified in Figure 17. "Cnt" has been set to the value of "cnt" in the NM template. Specificiation of the "cnt" value in the NM template was discussed above and shown in Figure 18.

The value of "constraint" is unset. "Constraint" is the only parameter that may be left unset. The meaning of "constraint" is discussed later and is not used in the previous examples.

6.7 Connections

POs communicate via channels. The accessibility of a channel from a PO instance is represented by a connection line in an NM template between the PO instance and the channel. NM template ports may serve as surrogate channels and PO instances may connect to NM template ports (see Figure 6). When an instance of the NM class is created within another NM class template the ports of the NM instance may be connected to channels in the new NM template (see Figure 5). Alternately the template ports of the new NM template may serve as surrogates producing any number of levels of indirection. The indirection of connections through NM ports does not effect the efficiency or cost of communication in any way.

There can be only a single connection from an instance port. Multiple connections may be made to a channel instance or template port.

Connection mode is entered by selecting the "CONNECT" command. A connection is made by selecting an NM or PO instance port, then selecting a channel instance or template port. If a bent connection line is desired then select points within the template background before selecting the channel instance or template port.

Disconnect mode is entered by selecting the "DISCONNECT" command. Selecting a connected instance port will delete the connection.

6.8 Generating Load Files

Once an NM template or PO template definition is completely defined within the graphical interface, load files may be generated. The "OUTPUT" command allows the user to select the target language and file name for a load file. Currently four target languages are supported. Concurrent Utah Scheme, DPOS Scheme, DPOS-DEBUG Scheme, and Multi-scheme (or Butterfly Scheme). DPOS Scheme and DPOS-DEBUG Scheme use the DPOS Scheme dialect described in this document.

7 Further DPOS Concepts

This section presents several DPOS concepts not discussed in earlier sections. The concepts presented here facilitate the management of groups of processes using recursive Network Modules and the dynamic allocation of processes. In previous example programs processes have been statically defined. The type and number of processes were determined prior to execution of the program. DPOS provides two mechanisms for management of dynamic process creation,

constraints and delays.

7.1 Constraints

Constraint conditions allow the creation of processes to be controlled at run time. A **constraint** condition is a test condition that constrains the creation of Network Modules and Process Objects. **Constraints** may be specified for nested NM and PO instances in Network Module class definitions. The **constraint** condition appears in the parameter menus of Process Object and Network Module instances (see Figure 22). If the **constraint** condition is unset then the NM or PO will automatically be created when the parent NM is created. If the **constraint** condition is set then the condition will be evaluated automatically when the parent NM is created. The **constrained** NM or PO instance will be created only if the evaluation returns non-nil. The **constraint** condition may use any values visible within the scope of the parent NM.

The use of constraint conditions allows the recursive and mutually recursive definition of Network Module classes.

7.2 Delays

In addition to **constraints**, PO and NM instances may be **delayed**. If a PO or NM instance is **delayed** then it is not created when the parent NM instance is created. Its creation is **delayed** and may or may not be carried out at some future time. The creation of **delayed** NM and PO instances is triggered by message traffic through one of its connecting channels. The trigger channel is called a **delay** channel. The first message access to a **delay** channel will cause the creation of the **delayed** instance.

If a PO or NM instance is to be **delayed** then it must have at least one connecting **delay** channel. The **delay** relationship between a channel and a PO or NM instance is specified within the parent Network Module definition as an attribute of the connection. A channel may be **a delay** channel for any number of connected instances.

The use of **delayed** instances allows the recursive and mutually recursive definition of Network Module classes.

The **delay** attribute of a channel is specified by using the "SET DELAYS" interface mode command. Then selecting connections to be **delayed**.

7.3 Channel Streams

In previous sections channels have been specified as single entities. DPOS channels may alternately be specified as streams. A DPOS **stream** channel is a delayed stream of channels. **A stream** channel is like **a list** of similar channels. It differs from a **list** in several ways:

1. The stream is of potentially infinite length.
2. The channel instances are **delayed** (not initially created).
3. The **stream** may be accessed only using stream accessor functions.
4. The first access of a stream element automatically triggers the creation of that element.

A number of stream accessor functions are provided:

1. **h-strm**: returns the first channel (head) of the stream.
2. **t-strm**: returns the remainder of the stream (following the head).
3. **ht-strm**: returns the second channel of the stream. This is equivalent to (h-strm (t-strm channel-strm)).
4. **htt-strm**: returns the third channel of the stream.
5. **httt-strm**: returns the fourth channel of the stream.

Stream channels are a convenient way to create multiple channels without having to worry about the number of channels actually needed. A channel instance may be specified to be a **stream** channel using the "SET STREAM" interface mode command. **A stream** channel may not be used as a **delay** channel.

Connections may be made to a **stream** channel just as to a non-stream channel. A Process Object may receive a **stream** channel as a parameter. A Process Object may only access a **stream** channel by using **stream** accessor functions. So, for example, a Process Object may send a message to the second channel of **a stream** channel of type **a_channel** by using the command **(send (ht-strm chan) data)**.

Alternately connections to a **stream** channels in Network Module class definitions may invoke accessor functions. This allows a connection to be made to

Figure 23: Instance Port Parameter Menu

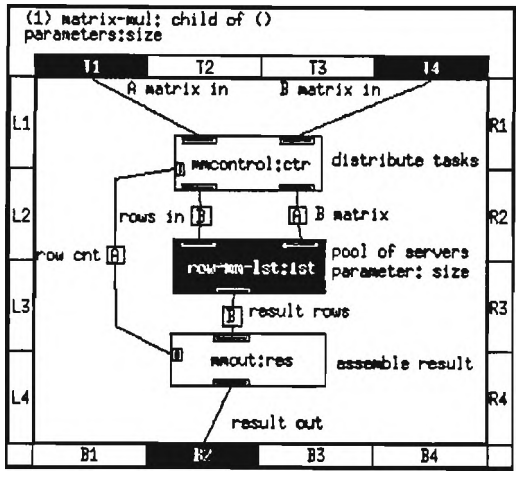

Figure 24: Matrix Multiplication Network Module

only one member of the **stream** channel. So a nested NM or PO instance would only see a single channel object as its parameter. The accessor function may be specified for a connection by using the "EDIT" interface function to edit the instance port parameter menu (see Figure 23). If the accessor function is left unset then the connection is made to the entire **stream** channel.

8 Further Examples

This section contains example programs using the features described in section 7.

8.1 Matrix Multiplication

This example program multiplies an L x M matrix (A) by a M x N matrix (B). The problem is typical of many numerical computations and other problems with very regular structure. The solution uses the same basic strategy as the example program in subsection 5.3. It uses a pool of servant processes to carry out similar computations.

Network Modules for the program are shown in Figures 24 and 25. In the program a control process of class **mmcontrol** first sends the B matrix to a channel labeled "B matrix" where it is read each of the **row-mm** class servant processes defined in **row-**

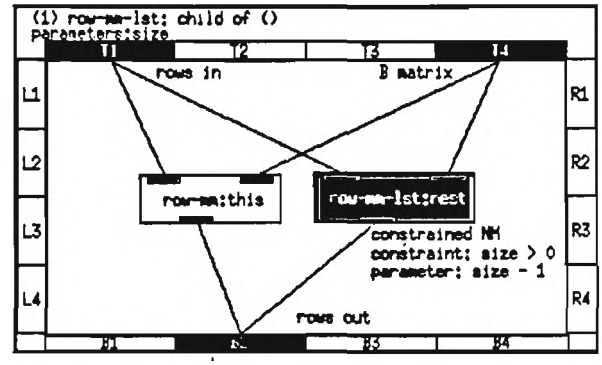

Figure 25: Server Process List Network Module

```
(define (row-mm-po ROWS-IN B-MATRIX RESULT-OUT)
 (let ((matb (chan-read B-MATRIX)))
  (do ((row (receive ROWS-IN) (receive ROWS-IN)))
      ((not row))
      (let ((index (car row))
            (x (cadr row))
            (y (caddr row))
            (rowvec (cadr (cddr row))))
       (send
         RESULT-OUT
         (list index (row-x-mat x y rowvec matb)))))))


(define (mmcontrol-po A-MAT-IN B-MAT-IN
                      ROWS-IN B-MATRIX ROW-CNT cnt)
  (let* ((matb (receive B-MAT-IN))
         (mata (receive A-MAT-IN))
         (x (vector-length mata))
         (y (vector-length matb))
         (z (vector-length (vector-ref matb 0))))
   (send B-MATRIX matb))
   (send ROW-CNT x)
   (do ((a 0 (+ a 1)))   ;send out the rows
     ((= a x))
     (send ROWS-IN (list a y z (vector-ref mata a))))
   (do ((a 0 (+ a 1)))   ;send out terminations
     ((= a cnt))
     (send ROWS-IN #f))))

(define (mmont-po RESIN RESOUT SIZEIN)
 (let ((size (receive SIZEIN))
       (resvec #f)
       (newres #f))
   (set! resvec (make-vector size))
   (do ((a 0 (+ a 1)))
     ((= a size))
     (set! newres (receive RESIN))
     (vector-set! resvec
                  (car newres) (cadr newres)))
   (send RESOUT resvec)))
```

Figure 26: Process Object Code

mm-lst:lst. It then distributes rows of matrix A to the servants on a first-come first-served basis through channel "rows in". Computed rows are sent to a collector process **mmout:res** which assembles the resulting matrix. Process Object **mmout:res** receives the number of rows from **mmcontrol:ctr** via channel "row cnt" to determine when it has received all resultant rows.

The list of servant processes is defined as a recursive Network Module in Figure 25. The number of servants is determined by defining a **constraint** condition on the nested recursive NM instance **row-mm-lst:rest**. The **constraint** condition "size > 0". If the result is non-nil then the recursive NM is created and passed the parameter "size - 1". The decrementing of size in successive nested instances limits number of servant processes.

The individual servant processes are of class **row-mm** and the source code is shown in Figure 26. The servant processes all share common access to the channels in the **matrix-mul** Network Module. The program maintains a balanced workload by taking advantage of shared channel "rows in" to allow the nondeterministic distribution of row vectors to servant processes. The channels marked "B" are buffered to relax the synchronization between processes as much as possible. The program uses asynchronous channel "B matrix" which buffers a single message and allows input operations **receive** with removal or **chan-read** without removal of the channel contents (see Figures 24, and 26). Using **chan-read** in this program implements a read only shared variable for the servant processes.

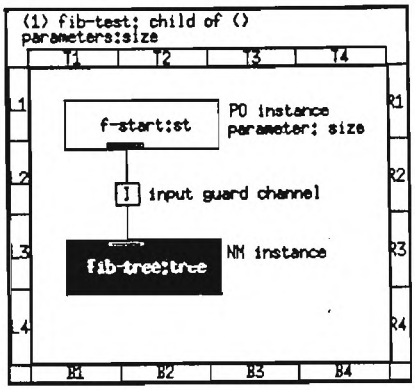

Figure 27: Fib-test Network Module

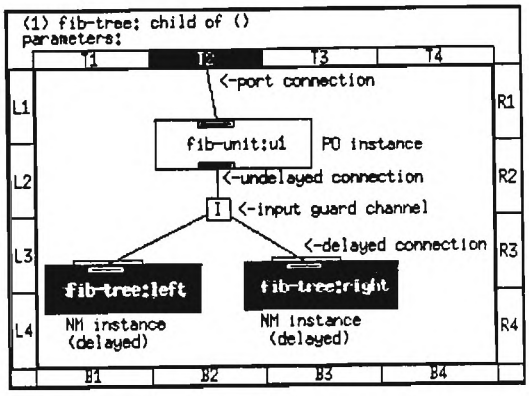

Figure 28: Fib-tree Network Module

8.2 Fibonacci Numbers

This program presents the naive recursive algorithm for computing the fibonacci sequence (see Figures 27, 28 and 29). The program includes two Process Object classes, a controller process of class **f-start** and worker processes of class **fib-unit**. The entire Process Object source listing is given in Figure 29. Process objects **f-start** and **fib-unit** use interface functions **f-start-po** and **fib-unit-po** respectively (see Figure 29).

The Network Module definitions are shown in Figures 27 and 28. Figure 27 shows the top-level Network Module. Figure 27 contains an **f-start** class Process Object called **f-start:st** and a **fib-tree** class Network Module called **fib-tree:tree**. The **fib-tree** Network Module is shown in Figure 28. It contains a **fib-unit** PO and two delayed **fib-tree** NMs. The block visible at the instance port of each nested **fib-tree** NM signifies that the connection is a delay con-

```
(define (f-start-po CHAN size)
 (write-guard CHAN ask size)
 (display
   (list 'answer (read-guard CHAN (list reply)))))

(define (fib n)
  (if (< n 3)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))

(define (fib-unit-po TOP BOT)
 (let ((size (cadr (read-guard TOP (list ask))))
       (grainsize 7))
  (if (< size grainsize)
   (write-guard TOP reply (fib size))
   (begin
    (write-guard BOT ask (- size 1))
    (write-guard BOT ask (- size 2))
    (let ((res1 (cadr (read-guard BOT (list reply))))
          (res2 (cadr (read-guard BOT (list reply)))))
     (write-guard TOP REPLY (+ res1 res2)))))))
```

Figure 29: Fibonacci Process Objects

nection and that the NM is delayed.

The only argument to the **Fib-test** NM is the value of **size** which determines the fibonacci number to be computed. **F-start:st** sends the seed **size** to its channel then reads the result when the computation is complete. **Fib-unit** process objects receive a **size** value from their top channel and if **size** is less than **grainsize** they do the fib calculation for that **size**. If **size** is greater than **grainsize** they send **size**-1 and **size**-2 to their bottom channel which triggers the creation of the delayed **fib-tree** Network Modules who do the work. The child processes then carry out the subcomputations and return the results.

This example demonstrates the use of guarded input channels for bi-directional communication. Message types used are **ask** and **reply**. The use of guard channels ensure that only appropriate message types are received. If guard channels are not used then a separate input and output channel would be required for both top and bottom connections to ensure that high-level race conditions do not occur.

8.3 Router Network

The router network program presented in this subsection demonstrates the use of a **stream** channel to implement a highly connected group of Process Objects (**routers**). The program concentrates on the communication aspect of DPOS programming. The sole work of **router** POs is to route messages.

The Network Modules for the program are shown in Figures 30 and 31. The **router-net** class NM in Figure 30 contains an **init-router** PO which sends messages to a list of **router** POs in **router-lst:lst**. A message is a list of forward addresses and commands. The **routers** simply forward messages among each other until an "OUTPUT" command, then send the tail of the message to **router-out:outer**. When all messages have been received, **Router-out** sends a termination message to the list of **routers**. The **router-lst** class NM in Figure 31 implements a list of **router** Process Objects using **constraints** as demonstrated in subsection 8.1.

The channel labeled "B" in Figure 30 is designated a **stream** channel (indicated by nested box) of buffer channels. This **stream** channel is connected to **router-lst:lst** through two ports. Within NM class **router-lst** the access to port "T4" connects all POs to the entire stream of channels. T1 is connected to the entire list in the first **router-lst**, to the tail of the list in the second, to the tail of the tail in the third and so on. Each successive **router** PO connects to the head of T1. So consecutive **router** POs connect through T1 to consecutive channels in the **stream**

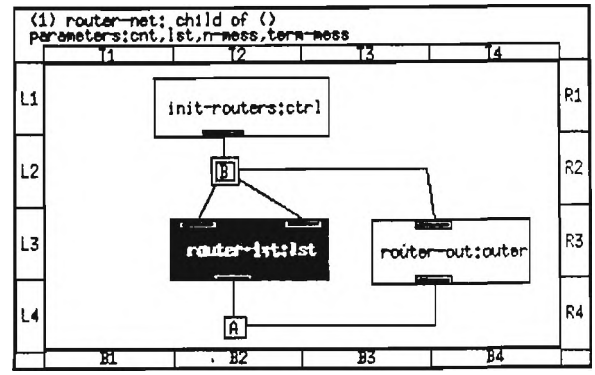

Figure 30: Router-net NM

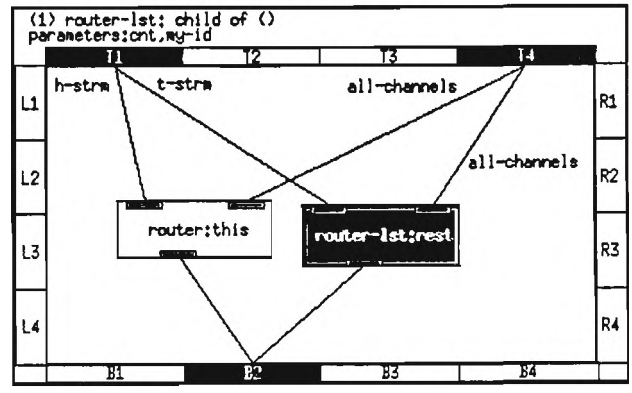

Figure 31: Router-lst NM

```
load channel_debug.scm
DEFS
load router-net.dp
load router-lst.dp
load init-routers.dp
load router-out.dp
load router.dp
BODIES
(init-debug 'router-net)
(let ((lst
       '((1 (a 2)(b 3)(output m1))
         (2 (a 1)(b 3)(c 2)(output m2))
         (0 (a 3)(b 2)(c 3)(output m3))
         (3 (a 2)(b 1)(c 0)(output m4))
         (1 (a 2)(output m5))
         (1 (a 2)(b 3)(output m6))))
      (end-mess1
       '((stop 1)(stop 2)(stop 3)(stop #f))))
  (router-net 4 lst 6 end-mess1))
ENDBODIES
```

Figure 32: Run File

```
(define (init-routers-po ALL-CH lst)
 (let ((inval #f)
       (next #f)
       (message #f))
  (do ((a lst (cdr a)))
      ((not a))
   (set! next (car a))
   (send-nth ALL-CH (car next) (cdr next)))))

(define (router-out-po ALL-CH OUT-CH n end-mess)
  (do ((a 0 (+ a 1)))
      ((= a n))
    (display (list (receive OUT-CH) 'received)))
  (send-nth ALL-CH 0 end-mess))

(define (router-po MY-CH ALL-CH OUT-CH id)
 (let ((inval #f)
       (next #f)
       (message #f)
       (done #f))
  (do ((a 0 (+ a 1)))
      (done)
   (set! inval (receive MY-CH))
   (set! message (caar inval))
   (set! next (cadr (car inval)))
   (if (equal? message 'output)
      (send OUT-CH (cadr (car inval)))
      (begin
       (if (equal? message 'stop) (set! done #t))
       (if next
         (send-nth ALL-CH next (cdr inval))))))
  (display (list 'router id 'stops))))

(define (send-nth CH-STRM n mess)
  (send
   (do ((a 0 (+ a 1))
(STR CH-STRM (t-strm STR)))
       ((= a n) (h-strm STR)))
   mess))
```

Figure 33: Process Objects

```
(m5 received)
(m4 received)
(m1 received)
(m3 received)
(m6 received)
(m2 received)
(router 0.000000 stops)
(router 1.000000 stops)
(router 2.000000 stops)
(router 3.000000 stops)
```

Figure 34: Program Output

channel.

Figure 33 shows the Process Object definitions for the program. In the **router-po** definition, "MY-CH" corresponds to the head of "T1" while "ALL-CH" corresponds to "T4". Each **router-po** receives messages from "MY-CH" and forwards messages to a specified channel in "ALL-CH" using the function **send-nth**. If **type** is 'stop then the PO terminates after forwarding. if **type** is 'output then the PO forwards the message to "OUTPUT" instead of "ALL-CH".

9 DPOS Scheme Interpreter

The DPOS Scheme interpreter is a parallel interpreter with simulated concurrency. The level of parallel simulation is below the level of DPOS parallel constructs. Also, the interperter inserts random timing delays at all possible synchronization points. This means that any synchronization condition that can arise on a parallel processor can be simulated by the interperter. If the duration of the computation is short, some possible synchronization conditions may not arise either in simulation or true concurrency.

There are several advantages to using the simulator for initial program development.

1. The turn around time is shorter because files do not have to be compiled and executable templates do not have to be generated.

2. Results are repeatable. Identical repeated runs will produce identical output.

3. Debug interpreter output may be immediately read by the DPOS graphical interface for debugging.

4. The interpreter runs in a familiar environment (unix).

5. The interpreter runs at speeds comparable to a uniprocessor version of CUS or Multischeme.

9.1 Using the DPOS Scheme Interpreter

There are two version of the DPOS Scheme interpreter, **dpos** and **debug**. **Dpos** reads files generated using the "DPOS-SCHEME" output format. **Debug** reads file generated using the "DPOS-DEBUG" output format.

The DPOS Scheme interpreter reads a single run file (see Figure 3) which must specify all top level

< Channel Definitions >
DEFS
< Load Files >
< Function Definitions >
BODIES
< Scheme Commands >
ENDBODIES < carriage return >

Figure 35: Run File Format

definitions for the program and must also specify a sequence of commands to be executed.

Run files may contain DPOS Scheme source code and also DPOS Scheme run file commands. Run file commands are not Scheme commands and do not use Scheme syntax. The DPOS Scheme run file commands cause files to be loaded and evaluated at top level and also separate the run file into blocks.

If the program uses concurrent features, the run file must load a DPOS channel file. Run files may load in Process Object and Network Module definition files (see sections 4.2 and 4.3). The load command has the form: "**load** filename". Load files produced with the DPOS graphical interface have the extension ".dp" added to them automatically. The run file has a standard form shown in figure 35.

9.2 Run Files for the standard Interpreter

Run files for the standard interpreter must meet the following requirements:

1. The channel definitions for the standard interpereter are in "channel.scm". This file must be loaded before the DEFS command in the run file.

2. All Network Module load files must be generated using the DPOS-Scheme output option in the graphical interface.

3. The BODIES section must meet the following requirements.

   (a) The BODIES section may invoke one or more Network Module definitions. These top level Network Modules may have parameter value arguments but must have no outer port connections.

   (b) Any number of sequential Scheme statements may be included.

9.3 Run Files for the Debugging Interpreter

Run files for the debugging interpreter must meet the following requirements:

1. The channel definitions for the debugging interpereter are in "channel_debug.scm". This file must be loaded before the DEFS command in the run file.

2. All Network Module Load files must be generated using the DPOS-DEBUG output option in the graphical interface.

3. The BODIES section must meet the following requirements.

   (a) The BODIES section must invoke at most a single Network Module definition. In addition this top level Network Module may have parameter value arguments but must have no outer port connections.

   (b) The first statement in the BODIES section must be (**init-debug** NM-name) where NM-name is the name of the Network Module definition that will be invoked. This does not invoke the NM, it merely tells the interpreter what template is to be the first process created.

   (c) Any number of sequential Scheme statements may be included after the **init-debug** statement.

   (d) The single Network Module invocation may occur at any time after the **init-debug** statement.

9.4 Running the Interpreter

The interpreter programs are called **dpos** and **debug**. The interperter output is to standard out. The command to run the interpreter is:

```
dpos -f run-file-name
```

or

```
debug -f run-file-name > tmp.file
```

Run-file-name is the name of a DPOS Scheme source file. The output from **debug** contains trace statements readable by the graphical interface. The output from **debug** may be sent to a file for later animation and debugging using the graphical interface.

10 DPOS Scheme Features

DPOS Scheme is a concurrent Scheme dialect. DPOS Scheme contains a subset of standard Scheme features and includes additional extended features for specifying concurrent program execution.

10.1 Deletions from Standard Scheme

1. Global variables are not defined.
   DPOS Scheme is a distributed memory language without global variables.
2. Standard forms and functions may not be reset by user programs.
3. The source file may cause other files to be loaded.
4. Strings are not supported. Symbols are automatically coerced into strings in commands requiring strings.
5. Continuations are not supported.

10.2 Standard DPOS Scheme Features

This section contains list of standard Scheme functions supported by DPOS Scheme. The syntax of each procedure or form is given. When several forms appear on a line the syntax is identical for all. When the semantics of some form or function differs from standard Scheme an explanation is given.

Special Forms:

1. (**define** (variables formals) body)
   define may only be used to define functions. DPOS Scheme does not support global variable definitions.
2. (**let** (bindings) body)
   Bindings have the form: (variable init).
3. (**letrec** (bindings) body)
   Bindings have the form: (variable init). Bindings are evaluated in the order given.
4. (**begin** body)
5. (**if** test consequent alternate)
6. (**cond** clause1 clause2...)
   Clauses have the form: (test expression).
7. (**lambda** (formals) body)
8. (do ((variable1 init1 step1)...) (test expression) command...)
9. (**while** test body) Test is evaluated. If the result is not #f then body is evaluated. This is repeated until test returns #f.

Standard Procedures and definitions:

1. (**equal?** obj1 obj2)
2. (**eq?** obj1 obj2), **eqv?** In DPOS Scheme **eq?** and **eqv?** are identical.
3. (**and** exp1 exp2...), **or**
4. (**not** exp1)
5. (**set!** variable expression)
6. (**apply** procedure list)
7. (**procedure?** expression)
8. #f
9. #t
10. (**delay** expression)
11. (**force** delayed-exp)

List Procedures:

1. (**car** exp), **cdr**, **cadr**, **cdar**, **cddr**
2. (**cons** exp1 exp2)
3. (**list** exp1 exp2 ...)
4. (**list-ref** list k)
5. (**list?** obj)
6. (**set-car!** pair obj), **set-cdr!**
7. (**list->vector** list)

Numeric Functions:

1. (< x1 x2) , >, <=, >=
2. (+ x1 x2 ...), *
3. (- x1 x2), /
4. (**remainder** x1 x2)
5. (**truncate** x)
6. (**number?** obj)

Vector Functions:

1. (**vector-ref** vector n)
2. (**vector-set!** vector n obj)
3. (**make-vector** n)
4. (**vector->list** vector)

5. (**vector?** obj)

6. (**vector-length** vector)

I/O Functions:

1. (**display** obj) or (**display** int obj), write, print

2. (**newline**)

3. (**read**) or (**read** int)

4. (**open-output-file** name)

5. (**open-input-file** name)

6. (**close-file** int)

7. (**close-output-file** int)

8. (**close-input-file** int)

Additional functions are supplied in a separate library "scheme_lib.scm".

1. (**length** list)

2. (**map** proc list)

3. (**append** list1 list2)

4. (**caaar** list), **caadr**...**cdddr**

5. (**for-each** proc list)

6. (**member** obj list), **memq**

7. (**assoc** obj list), **assq**

8. (**quotient** n1 n2)

9. (**max** n1 n2), **min**

10.3 DPOS Scheme Extensions

DPOS Scheme incorporates several features to support parallel program definitions. The features are intended to support the definition of DPOS metalanguage programs. When using the DPOS metalanguage some of the extended language features are specified by the programmer and some are typically specified only in source code generated by the DPOS graphical programming interface.

Process Object Functions:

1. (**send** channel data)
   Send a message to a channel. The channel must be either asynchronous or buffer class.

2. (**receive** channel)
   Receive a message from a channel and remove the contents. The channel must be either asynchronous or buffer class.

3. (**chan-read** channel)
   Receive a message from a channel and leave the contents intact. The channel must be asynchronous class.

4. (**write-g** channel type message)
   Write a message of type "type" to a channel. The channel must be guarded input type.

5. (**read-g** channel type-list)
   Receive a message of any one of "type-list" types from a channel. The channel must be guarded input class.

6. (**h-strm** streamchannel)
   Returns the first channel of a stream channel.

7. (**ht-strm** streamchannel)
   Returns the second channel of a stream channel.

8. (**htt-strm** streamchannel)
   Returns the third channel of a stream channel.

9. (**httt-strm** streamchannel)
   Returns the fourth channel of a stream channel.

10. (**t-strm** streamchannel)
    Returns the tail of a stream channel.

10.4 Source File Commands

Source File Commands:

1. **DEFS**
   Precedes the first **define** or load of a user generated definition file.

2. **BODIES**
   Separates defines and loaded file commands. From execution commands.

3. **load** name Used before **DEFS** to load the channel definitions (either "channel.scm" or "channel_debug.scm"). Used between **DEFS** and **BODIES** to load user generated definition files.

4. **ENDBODIES**
   Follows the last execution command. **This must be followed by a carriage return.**

10.5 Interface Specified Functions and Commands

This subsections lists functions and commands generated by the DPOS graphical interface. These functions are not intended to be specified by programmers. These functions are also implementation specific for the DPOS Scheme simulator. If used by the programmer they will generate programs that are not useable by the graphical interface for editing and debugging. The argument lists shown are specific for

the debugging interface version of DPOS Scheme and vary from those for regular DPOS Scheme.

Interface generated functions and commands:

1. (**system-call** parameter-list) This is the hook into the system. Several useful operations are available using the listed argument lists.
   (a) 'random :generate a random integer 0..99
   (b) 'rand n :generate a random float 0..n
   (c) 'wait n :delay execution n steps
2. (**init-debug** name)
3. (**a_channel** name index id)
   Create an asynchronous channel generator.
4. (**b_channel** name size index id)
   Create a buffer channel generator.
5. (**i_channel** name size guard-cnt id)
   Create a guarded input channel generator.
6. (**new-process** procedure arglist name id)
   Create a new process.
7. (**delay-channel** channelform delay-lst)
   Create a delay channel.
8. (**new-port** location channel)
   Initiate the debugging interface for a port.
9. (**dpos-id**)
   Generate an identifier for a channel. or object instance.
10. (**streamof** thunk)
    Generate a stream of channels.
11. (**activate-delay-forms** list)
    Evaluate the list of delayed processes for a delayed channel.

11 Portability Across Target Languages

The DPOS graphical interface generates source code in four target languages. This means that an NM class definition created using the interface is portable across the four languages. The Process Objec source code, however, is specified by programmers using a text editor. Scheme is the base language for all four languages. Each of the dialects contains a subset of the essential features of sequential Scheme. Each dialect also adds some extensions including sequential commands and parallel commands.

A subset of DPOS Scheme may be used which is compatible across all four languages. The compatible subset of features includes all features listed in subsections 10.2 and 10.3 with a few exceptions.

1. **list?** is called **pair?** in CUS.
2. **while** is supported only in DPOS Scheme.
3. **print** is not defined in CUS.
4. **close-file** is supported only in DPOS Scheme.
5. **force** and **delay** may vary semantically in CUS.
6. Some errors in DPOS Scheme produce an error message but return the value #f allowing the program to continue.