Software Engineering Institute

Technical Report

ESD-TR-86-212

CMU/SEI-66-TR-6

December 1986

Version 0.1 August 30, 1986

Version 0.2 November 15, 1986

The Heterogeneous Machine Simulator

Robert G. Stockton

Carnegie Mellon University Pittsburgh, Pennsylvania 15213

This research is carried out jointly by the Software Engineering Institute, a Federally Funded Research and Development Center, sponsored by the Deparbnent of Defense , and by the Department of Computer Sciince. sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-64-K-1520.

Table of Contents

1. Introduction 1
2. Hardware Configuration 2
    2.1. Processors 2
    2.2. The Central Switch 2
    2.3. The Scheduler 3
3. Software Configuration 4
    3.1. The Task Library 4
    3.2. Processes 5
    3.3. Queues 6
    3.4. Special Processes 6
4. Simulating a Program 7
5. Assumptions 8
6. Appendix I -- Example 9
    6.1. Simulation Input 9
    6.2. Simulation Output 12
7. Appendix II -- Simulator Commands 15

The Heterogeneous Machine Simulator

Abstract

The heterogeneous machine simulator is a program which attempts to simulate the proposed hardware for the heterogeneous machine at a high level, along with the low level programming abstractions which have been proposed. This will, hopefully, provide: 1) a reasonable basis for programmers to evaluate application designs in the absence of the actual machine; and 2) a testbed for designers to experiment with various reconfigurations which might be difficult to perform on the machine itself. This document presents a basic description of the system, and presents an example of how a simulation may be run.

1. Introduction

The heterogeneous machine simulator is a program which attempts to simulate the proposed hardware for the heterogeneous machine at a high level, along with the low level programming abstractions which have been proposed. This will, hopefully, provide: 1) a reasonable basis for programmers to evaluate application designs in the absence of the actual machine; and 2) a testbed for designers to experiment with various reconfigurations which might be difficult to perform on the machine itself. Since neither the design of the machine or the programming abstractions are firmly fixed at present, the program has been written in such a way that it should be fairly easy to modify in accordance with any design changes.

The reader of this document is presumed to have available a copy of the Language Reference Manual [l] which provides more complete explanations for many of the abstractions described here. There are, however, some minor syntactic differences between the descriptions given in that document and implementations within the simulator. An attempt will be made to point out all such differences.

2. Hardware Configuration

Within the basic design of the Heterogeneous Machine, there is a reasonable amount of flexibility in the exact configuration of hardware. This section describes the various ways in which the user of the simulator may modify the configuration of the simulated machine. It also attempts to reveal various assumptions concerning the hardware which have been wired into the simulator. It is important to note that, although these assumptions may not be modified at the time the simulator is run, they will probably not require massive effort to modify within the simulator. It is expected that several of these will need to be changed at some point.

In general, the user will always work with a certain standard configuration, and therefore an initialization file (HetSim.init) is loaded whenever the simulator is run. This configuration may then be further modified by the user after the file has been loaded. This file may, in fact, contain any simulator commands the user desires. It may, for example, also contain a command to read in a standard task library (see Section 3.1 below).

2.1. Processors

The simulator does not have any particular configuration of hardware processors wired into it. Instead the user may declare a collection of processors of varying types. At present, a processor is described merely by a name and a processor type (i.e. “warp”, “sun”, etc.).

When a processor is declared, it also has a communications processor, or buffer, associated. with it by the simulator. The current system assumes that each buffer has exactly one processor associated with it. In addition, it assumes that the buffer may both receive data from and submit data to the processor, and that the buffer may temporarily store data before sending it to its final destination. This buffer will also be used to implement some simple data transformations and several special operations (i.e. merges, deals, and broadcasts). These operations will be described later in this document.

Example:

```
> processor
Name: procl
Processor Type: sun
```

2.2. The Central Switch

It is assumed that there is only one (logical) switch in the heterogeneous machine, and therefore it need not be declared. However, it is possible to modify the default assumptions concerning the speed of the switch. The speed is specified in terms of the overhead time for each message (in milliseconds) and the data transmission time per bit (also in milliseconds). The default speed is 0.001 milliseconds overhead, and 0.0002 milliseconds per bit. In addition, it is possible to request information concerning switch traffic during a program simulation.

The simulation of the switch is based upon the assumption that there will be two classes of priorities for messages: a high priority for control messages, and a lower priority for data transmissions; and that each message which is submitted for transmission is sent as a single non-interruptable package, and that no other message may be sent while it is in transit. Like most of the assumptions made in the simulator, this may easily be modified in the future if it is incorrect.

Example:

```
> Switch
Message overhead (ms): 0.005
Time per bit (ms): 0.001
```

2.3. The Scheduler

At present, the simulator does not deal explicitly with any processor which is responsible for system scheduling. Most of the operations which would be performed by the scheduler processor are available to the user of the simulator, and therefore this user may be considered to be playing the part of the scheduler. In the future when the operation of the proposed scheduler have been defined more precisely, the simulator may be modified so that a scheduler drives the current version of the simulator, and the user merely sets up the operation of the scheduler.

3. Software Configuration

The designer of a software system will typically begin with a single hardware configuration which will be loaded at start-up time. His task is then to produce descriptions for a collection of software components to be run on this hardware, and to specify the interconnections between them which are required to produce a coherent program. These are specified in terms of a variety of abstractions, which will be described in this section.

3.1. The Task Library

Before a user may simulate a heterogeneous machine program, descriptions must exist for the individual tasks that will constitute the program, as well as the types of data which are to be manipulated by the program. These descriptions are stored in the task library, which may be read to or from a disk file as needed. Presumably the user of the simulator will start with a library containing the standard tasks which are available as utilities, and will then add descriptions for a number of tasks which are to be specially written for the system being simulated.

At present, the simulator provides fairly simple facilities for creating and modifying task descriptions. A user may create a basic task description, consisting of a task name and a list of input and output *ports* (named, typed IO channels). He may then specify additional information concerning the task in the form of attributes, which consist of named strings which provide various sorts of informations about the task. The two attributes which are of primary interest in the simulator are the "processor" and "timing" attributes. The first of these is used to select a version of a task which is appropriate for the processor it is to run upon, and the second provides a general description of the behavior of the tasks operation which is used to simulate the tasks execution.

> COMMENT -- *This is a slight departure from the description in the language reference manual, which presents the timing information as an entity separate from the Attributes. The simulator also associates a special meaning with the 'LoadTime" attribute, which specifies the expected time required to download the code for the task to the processor. By default this is 0.*

Attributes for a task may be added, deleted, or modified by the user at any point. Facilities also exist to select tasks from the library based upon the name or upon task attributes.

Every data type which appears in a port description must be described to the simulator. At present, the simulator supports two varieties of types. Simple types are described merely in terms of their names and their sizes (in bits; the size of an array type is taken to be the element size times the number of elements). Union types represent a class of data types, all of which may be handled in the same way. One type is said to satisfy another if all of the simple types which constitute the first are also present in the second. The system provides facilities for creation or deletion of both simple and union types.

Examples:

```
> Union-Type Number
Components: real, integer
> Create-Task print
In Ports: Print-In, integer
Out_Ports:
> Set processor="sun"
> set timing="(Print_In [3,4.53)+"
> set author="hqb"
```

3.2. Processes

Since a task is fairly useless if it merely sits in a library, there must be some way for the code corresponding to a task description to be loaded onto a processor and executed. In the simulator, there exist facilities to select a task based upon its name, attributes, and the number and type of ports associated with it, and instantiate it on a given processor as a process. The ports corresponding to this process (which have the same types as the associated task, but may have different names) may then be connected to ports belonging to other processes in order to form a complete program. At present, the simulator assumes that only one process will be active on a processor at any given time.

The operation of a process is simulated based upon the "timing" attribute of the associated task. This template is assumed to fairly precisely characterize the I/O behavior of the process when run upon the processor it was written for. All of the internal operations of the process are assumed to be reflected in the delays specified between successive I/O events. The process may have multiple threads of control, and certain sequences of operations may be iterated a certain number of times (or indefinitely). Thus the following template

```
"( [3,5] (([2,4] out_port.integer)+lO || ([3.5,*] out_port.real)+lO))+"
```

describes a process which repeatedly: waits between 3 and 8 milliseconds, and then outputs 10 integers at intervals of 2 to 6 milliseconds at the same time that it outputs 10 reals at intervals exceeding 3.5 milliseconds (averaging 7 milliseconds), waiting for both sets of output operations to complete before starting the next iteration, The fact that output operations are being performed is inferred from the fact that outgort is an output port. Presumably out_port accepts a union type that includes both integers and real numbers. In this example, there are no input ports.

COMMENT -- The syntax for timing expressions varies somewhat from the language reference manual:

- *Type specifiers for output operations have been added, while the operation specifier has bean dropped.*
- *Time windows in queue operations are not allowed. Time windows in delay operations consist of relative times. Notice that the word "delay" is omitted before the time window.*
- *Guards of the form when, before, during, and after are not implemented. Only the repeat guard is implemented.*
- *The default time unit is the millisecond.*

Using the correct syntax from the language reference manual, the example above would be written as:

```
loop delay[0.003,0.008]
   (repeat 10 => (delay[0.002,0.006] out_port ||
    repeat 10 => (delay[0.0035,   l ] out_port))
```

Examples:

```
> Find print                    --| Creates candidate list
Requirements: Author="hqb"
> assign                       --| Selects one candidate from list
Processor: proc1
Process name: printprocess
In_Ports: inport,integer
Out_Ports:
```

3.3. Queues

Processes running on different processors exchange data by passing it through queues. The simulator views a queue as being a connection between two ports, with an optional upper bound upon its size. The assumption is made that data being sent through the queue may be temporarily stored in the buffer of either the input or the output processor, and that it is transmitted from one buffer to the other by means of the central crossbar switch, which is also used to transfer control messages in order to insure that both ends of the queue are consistent with each other. For convenience in reference, each queue within the simulator is given a unique name.

The user may specify data transformations to be performed upon data submitted to a queue. This transformation is described in terms of the input and output types (which may be union types) and the amount of time (milliseconds per bit of the input type) which is required for the buffer to perform the conversion. If the type conversion is too complex for the buffer to handle (or more efficiency is required), the conversion must instead be written as a task and run as a normal process. Several transformations may be specified on a single queue (corresponding to different members of a union type), but each simple type which may be produced by the feeding port must either satisfy the type of the consuming port, or satisfy a type conversion which produces a type which satisfies the consuming port.

In addition, it is possible during a program simulation to request the current size and average size of any queue.

Examples:

```
--| assume sort2 yields reals
> Connect -bound=20 printqueue, sort2.out, print_process.inport
Created with name printqueue
The following types are still unmatched:
    real
> Convert printqueue, real, integer
Time per bit (ms): 0.01
```

3.4. Special Processes

In order to increase the flexibility of the heterogeneous machine, a set of special operations have been proposed which would be implemented by the buffer processors. These operations: Merge, Deal, and Broadcast, appear similar to processes in that they consist of a collection of input and output ports connected via queues to ports in other processes. These special operations are each declared by a separate command within the simulator which accepts lists of input and output ports and a mode which further describes the process's operation, creates the desired special process, and associates it with the appropriate buffer. The assumption is made in simulating these special tasks that the data movement required to implement these operations within the buffers will take negligible time. It may be that this assumption is unreasonable, but it may easily be corrected if necessary.

Example:

```
> Broadcast
Out Port: generator.outport
In ports(s): sortl.in, print.inport -bound=20, dev_null.trashbin
--| The queue connecting the broadcast process to print.inport will be
--| limited to 20 elements. The process will be implemented upon the
--| same buffer as generator.outport
```

4. Simulating a Program

After the description of a complete program has been provided to the simulator, the user may tell the simulator to start the processes running and view the resulting execution. The internal operation of each process is simulated based upon the timing attribute (as described in Section 3.2), and the flow of data between processes is simulated in terms of the high level machine description and the abstractions described above. The system provides a screen oriented display of the operation of each process (in terms of a timing template with the current operation highlighted) and histograms showing the size of each queue. It also displays a running log of operations performed by various components of the system, which should thoroughly characterize the operation of the system. This log is also written to a file so that it may be studied in more detail later. (It is possible to turn off the screen display of the log, and in later versions of the simulator it will probably be possible to request that only selected classes of operations be displayed.)

The simulator is currently set to run the simulation until a given amount of virtual time has passed in the simulation (or until all processes have run to completion). When this time limit has been exceeded, the simulation stops and the user may either continue for another increment, or restart the simulator at time 0. Alternatively, he may clear the current simulation and set up a completely different one. In the future, this may by modified so that the user can also step through the simulation one simulator event at a time.

5. Assumptions

This section contains a brief (and hopefully somewhat complete) description of the assumptions made by the simulator, along with analyses of the difficulty of correcting any invalid assumptions. Most of these have been described elsewhere in this document.

- The speed of the switch is 0.001 milliseconds overhead, and 0.0002 milliseconds per bit. This is easily modifiable via a simulator command.
- Messages will be sent through the switch in two priority classes: Control and Data. This assumption (and the next) are isolated within the simulator code to allow easy modification.
- Each message is non-interruptable, and blocks other messages while it is being transmitted.
- Each buffer is associated with exactly one processor. This is easily modifiable within the simulator code.
- Buffers may handle both input and output data, and have a unlimited amount of storage available. The first assumption is more pervasive than the previous ones, but not impossible to modify. The memory requirement, on the other hand, may be softened considerably by specifying bounds on all queues in the application.
- The buffer's internal operation is fast enough that Merges, Deals, and Broadcasts may be implemented upon the buffer with negligible delay. Processing delays may easily be accounted for within the code, but the assumption that these special processes may be handled by the buffers is fairly deeply rooted.
- A fairly large class of simple data transformations may be performed by the buffer processor. If this isn't true, then this capability of the simulator may simply be ignored.
- The timing descriptions described are adequate to characterize the operation of a typical task. Extensions to these templates are certainly possible, but we do not yet have sufficient knowledge of the applications to determine what would be useful.

6. Appendix I -- Example

This example in this appendix illustrates the specification of a very simple application. The application consists of four processes and two special processes arranged as in figure 1. The first part of the appendix is a transcription of the interaction between the user and the simulator command interpreter. The second pat-l of the appendix is an abridged transcript of the simulator output, tracing the execution of the tasks and queue operations.

The transcripts have been annotated with comments to increase their readability. Comments are enclosed within the characters "--" and the end of the line and are not part of the user input or simulator output.

6.1. Simulation Input

The simulator prompts the user for commands by typing the character ">". All commands have the format "command-name parameters", but the user can elect to type only the command name, in which case the simulator will prompt the user for the missing parameters. See the following Appendix for a complete list of the available commands.

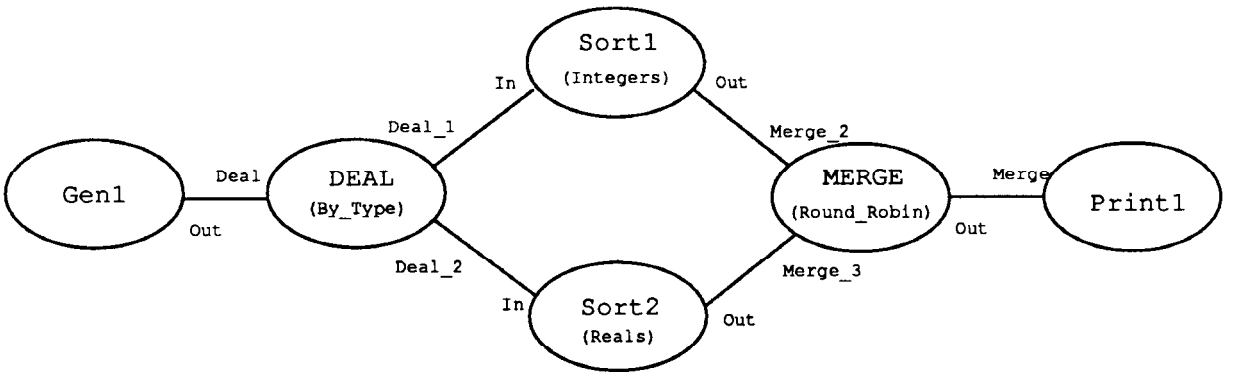
```
% hetsim
> processor
Name:pl
Processor Type: perq
> processor p2,sun
> processor p3,perq
> processor p4,perq
>  switch
Message overhead (ms): 0.001
Time per bit (ms): 0.0002

> read lib
>  library        --| brings up the entire library as the list to be
>  print-list     --| printed by the print list
Task  sort
  Ports:
    input:  IN  integer:
    output:  OUT  integer;
  Attributes:
    loadtime = "12"
    timing= "(input#lO [O,lO] output#lO)+"
    processor = "perq"

Task generator
  Ports:
    out :  OUT  integer;
  Attributes:
    timing= "([0,5] out)+"
    processor = "sun"

Task  print
  Ports:
    in:  IN  integer;
  Attributes:
    processor = "perq"
    timing= "(in [5,*])+"
```

```
Timing expressions
Genl:   (([0,4]  out.integer)+10 I I  ([0,5]  out.real)+10)+
Sortl:  (in#10  [0,10]  out#10)+
Sort2:  (in#10  [6,*]  out#10)+
Print:  (in [2.5,3.5])+
```



(All queues have limits of 10 elements)

Figure 1: Process Queue Graph

```
Task print
  Ports:
    in: IN number;
  Attributes:
    processor = "perq"
    author = "rgs"
    timing= "(in [2.5,3.5])+"

Task sort
  Ports:
    in: IN real;
    out: OUT real;
  Attributes:
    processor = "perq"
    timing= "(in#lO [6,*] out#lO)+"
    loadtime = "10"

> print-type number
Compound - components: real integer
           simple components: real integer

> create-task
Name: generator
In-Ports:
Out-Ports: out
Type: number
> set processor="sun"
> set timing="(([0,4] out.integer)+lO || ([0,5] out.real)+lO)+"
> set loadtime=5

> print-task
Task generator
  Ports:
    out : OUT number;
  Attributes:
    loadtime = "5"
    timing= "(([0,4] out.integer)+lO || ([0,5] out.real)+lO)+"
    processor = "sun"

> find sort                      --| locate sort tasks for the upcoming
 2 tasks found.                  --| assign
Requirements ('att=val,...'): processor=perq
Requirements ('att=val, . . . ') :
> assign
Processor: p1
Process name: sort1
In-Ports: in,integer            --| Disambiguation is done here on the
Out-Ports: out,integer          --| basis of port type

> find generator
 2 tasks found.
Requirements ('att=val, . . . ') :
> assign p2,genl                 --| This implicit requires that
In-Ports:                       --| the task can run on processor p2
Out Ports: out,number           --| (i.e. "require processor=sun")
```

```
> find sort
 2 tasks found.
Requirements ('att=val, . . . ') :
> assign p3,sort2
In Ports: in,real
Out_Ports: out,real

> find print
 2 tasks found.
Requirements ('att=val,...') : author=rgs
Requirements ('att=val, . . . ') :
> assign p4,printl
In Ports: in,number
Out_Ports:

--| The following commands implicitly create the queues Deal, Deal-l,
--| Deal_2, Merge, Merge_2, and Merge_3.
> deal
Out Port: genl.out -boundary=10     --| Port which will feed queue Deal
Mode: [Round_Robin] By_Type
In port(s): sortl.in -bound=lO,sort2.in -bound=10
                                    --| Fed by queues Deal-1 and Deal-2
> merge printl.in -bound=lO,Round_Robin
                                    --| Port to be fed by queue Merge
Out port(s): sortl.out -bound=lO,sort2.out -bound=10
                                    --| Feed queues Merge-2 and Merge-3

> verbose
Verbosity now set to false

> run 40
  --| screen oriented display omitted

> statistics
Control Messages:          23 messages        0.002 ms average delay
Data Messages:             59 messages        0.000 ms average delay
                                                66 bits average size
Switch was idle 99% of the time.
Queue Deal_1 had an average size of 3.1
Queue Deal_2 had an average size of 4.2
Queue Merge had an average size of 4.4
Queue Merge_2 had an average size of 3.3
Queue Merge_3 had an average size of 1.1
Queue Deal had an average size of 0.0
  quit
%
```

6.2. Simulation Output

An annotated segment of a simulator log follows:

```
—| The process "Sortl" writes 10 integers to its port "out", which
—| are sent through queue Merge-2 to the merge process feeding
—| print. Sort1 then immediately requests another 10 integers from
—| its input port "in".
25.945: Simulator - Processing event in class PROCESSOR
25.945: Queue Merge_2 - receiving data from sortl.out
25.945: Queue Merge_2 - Size now 10
25.945: Switch - Accepting 10 integers from out destined for Merge-In1
25.945: Switch - Transmitting 10 integers.
25.945: Process sort1 - finished sending data
25.945: Process sort1 - Generating successive events. 1 events generated.
25.945: Simulator - Processing event in class PROCESSOR
25.945: Queue Deal_1 - Processing request for data from sortl.in
25.945: Queue Deal_l - Delaying due to insufficient data.

——| The 10 integers sent through queue Merge_2, arrive at the other
——| end of the switch and are immediately processed by the merge
——| process implemented by buffer 4, which passes the first of these
——| integers on to the input of process "printl"
26.595: Simulator - Processing event in class SWITCH
26.595: Port Merge_In1 - accepting transmission of 10 integers from queue
                                                                Merge-2
26.595: Queue Merge - receiving data from Merge-Out
26.595: Queue Merge - Size now 1
26.595: Buffer 4 - Transferring 1 integers from Merge-Out to in.
26.595: Switch - Accepting control message from Merge_In1 to out.
26.595: Switch - Transmitting control message.
26.595: Simulator - Processing event in class SWITCH
26.595: Port in - accepting transmission of 1 integers from queue Merge
26.595: Queue Merge - Processing request for data from printl.in
26.595: Queue Merge - Size now 0

--| Process printl, which has been waiting for something to arrive in
--| its input port, accepts it immediately and prepares to munch on
--| it for a while.
26.595: Process print1 - receiving 1 integers
26.595: Process print1 - finished receiving data
26.595: Process print1 - Generating successive events. 1 events generated
26.595: Buffer 4 - Transmitting control message from in to Merge-Out.

--| The merge process is informed that the data it sent has been
--| consumed, sends a message back to port Sortl.out, informing it
--| that there is now an empty space in the queue. Since there was
--| no data pending on the port, this does not trigger any action.
26.595: Simulator - Processing event in class SWITCH
26.595: Port Merge_Out - Processing control message
26.595: Port Merge_Out- Recording consumption of 1 items from queue Merge
26.605: Simulator - Processing event in class SWITCH
26.605: Port out - Processing control message
26.605: Port out Recording consumption of 1 items from queue Merge_2
```

```
--| Process genl generates an item of type "real", which is passed
--| to the deal process implemented by the processors buffer. The
--| deal process passes the data straight to queue Deal2 (which is
--| where all reals are sent), which transmits the data through the
--| switch to port "Sort2.in".
27.126: Simulator - Processing event in class PROCESSOR
27.126: Queue Deal - receiving data from genl.out
27.126: Queue Deal - Size now 1
27.126: Buffer 2 - Transferring 1 reals from out to Deal_In.
27.126: Process genl - finished sending data
27.126: Process genl - Generating successive events.  1 events generated.
27.126: Simulator - Processing event in class SWITCH
27.126: Port Deal_In - accepting transmission of 1 reals from queue Deal
27.126: Queue Deal_2 - receiving data from Deal_Out2
27.126: Queue Deal_2 - Size now 9
27.126: Switch - Accepting 1 reals from Deal_Out2 destined for in.
27.126: Switch - Transmitting 1 reals.
27.126: Buffer 2 - Transmitting control message from Deal_In to out.
27.126: Simulator - Processing event in class SWITCH
27.126: Port out - Processing control message
27.126: Port out - Recording consumption of 1 items from queue Deal
27.264: Simulator - Processing event in class SWITCH
27.264: Port in - accepting transmission of 1 reals from queue Deal_2

--| Process Print1 finishes munging the last piece of data it got,
--| and asks for a new one.  Since the merge process is waiting for
--| the next data item to come from Sort2, Print1 is out of luck.
31.177: Simulator - Processing event in class PROCESSOR
31.177: Queue Merge  Processing request for data from printl.in
31.177: Queue Merge- Delaying due to insufficient data.
```

7. Appendix II - Simulator Commands

Commands consist of a keyword followed by a series of comma separated fields. (Fields may contain commas if they are enclosed in double quotes.) Keywords may be abbreviated to any unique prefix. Any fields which are not supplied will be prompted for, and any fields which are left blank will have defaults provided. Switches may follow any argument and take the for "-Switch=Value".m

ASSIGN Processor, Process-Name In_Ports Out_Ports
Selects a task from the current list (see FIND) which matches the port lists and instantiates it as a process, giving it the supplied name. tl then assigns the new task to the given processor (Note. that the list of out ports will have to be specified on its own input line).

ATTRIBUTE Attribute
Prints the value of the given attribute for the current task (see SELECT-TASK).

BROADCAST Out_Port -BOUNDARY=bound, <In_Port -BOUNDARY=bound, . ..>
Creates a broadcast special process connecting the given out port to all of the given in ports. Queues are automatically created with the boundaries given (or unbounded if no switches are supplied).

CONNECT Queue_Name, Out_Port In_Port -BOUNDARY=bound
Creates a queue between the given ports with an optional bound on the queue size.

CONVERT Queue, In-Type, Out-Type Conversion_Time
Specifies a type conversio which is to be carried out upon the contents of the given queue. All elements of the given input type will be converted to the given output time, taking the given amount of time in milliseconds per bit of input data. A list of types which are still unmatched will be printed.

CREATE-TASK Name, In_Port_List Out-Port-List
Creates a task and adds it to the current task library In addition, it selects the new task as the current task (see SELECT-TASK) Note that.the list of out ports will have to be specified on its own input line.

DEAL Out_Port -BOUNDARY=bound, <In_Port -BOUNDARY=bound, . ..>
Creates a deal special process connecting the given out port to all of the given in ports. Queues are automatically created with the boundaries given (or unbounded if no switches are supplied).

DEBUG Toggle printing of debugging messages.

DELETE Object Deletes processors, queues, or types (searching for them in that order).

FIND Task-Name ,Attribute=Value . . .
Locates all tasks in the library which have the given name and attribute values, and places them in the current task list where they may be operated upon by other commands.

HELP [* Command | ALL]
Help provides information on the simulator or on specific commands If the argument . i s then general information is printed "ALL" will. print information on all commands, or a single command name can be given and information on that command will be printed.

LIBRARY Sets the current task list to contain the entire program library.

LIST-TYPES Print the names of all types currently defined in the program library.

LOAD Filename Read a list of commands from a file and execute them.

MERGE In_Port -BOUNDARY=bound, <Out_Port- Boundary=bound,
Creates a merge special process connecting all of the given in ports to the given out port. Queues are automatically created with the boundaries given (or unbounded if no switches are supplied).

PRINT-TASK Print a readable description of the current task. (See SELECT-TASK)

PRINT-LIST Print readable descriptions of all tasks in the current list. (See FIND)

PRINT-TYPE Type-Name
Print a description of a single type.

PROCESSOR Name, Processor_Type
Create a new processor of the given type with the given name.

QUIT Exits the program, writing out log info to "test_log". Does not write out any changes which might have been made to the program library.

READ Filename Read the contents of a program library from the given file.

REMOVE-TASK Causes the current task to be eliminated from the program library. (See SELECT-TASK)

REQUIRE Attribute=Value
Removes all tasks which do not have the given attribute value from the current task list (see FIND).

RESET Returns the simulator to its original state, leaving only the program library.

RESTART Time Same as RUN except that it resets the clock and the program state back to the initial values.

RUN Time Start the simulator running for Time milliseconds. After each simulator step, the current state of each process and queue is displayed. In addition, if the Verbose switch is on, each I/O action is displayed. (A complete log of these actions will be written to the "test_log" file in any case.)

SATISFIES Type1 , Type2
States whether or not every component type of Type1 is contained in Type2.

SELECT-TASK Task-Number
Takes the Nth element of the current task list and selects it as the current task, so that it may be operated on by other commands.

SET Attribute=Value
Sets the value of the given attribute for the current task. (See SELECT-TASK)

STATISTICS Print a random set of statistics about the current simulator run.

SWITCH_TIME Message Time, Time_per Bit
Modifies the characteristics of the switch. Each message will take Message Time + Time_per_Bit • Message-Size milliseconds to transmit (after it reaches the switch).

TYPE_DEF Name, Size_in_Bits
Creates the given simple type in the program library.

UNION_DEF Name, <Component, . ..>
Creates the given union type with the list of component types given.

UNSET Attribute Removes the definition of the given attribute from the current task. (See SELECT-TASK)

VERBOSE Toggles verbosity. If on, then information concerning activity will be continuously printed as the simulator is running.

WRITE Filename Write the current contents of the program library to the given file.

References

[1] **M.R. Barbacci and J.M. Wing.**
Durra: A Task-level Description Language. (in process)
Technical Report , Software Engineering Institute, Carnegie Mellon University, 1986.