The Spring Simulation Testbed - V2
Programmer's Guide

Elizabeth Gene

May 1990

Contents

1 Overview **2**

2 Configuration Interface **2**

2.1 Makefile . . . 3

2.2 Files . . . 3

2.3 Data Structures . . . 3

2.4 Notes . . . 4

2.5 Modifications . . . 6

2.5.1 Adding a New Entry . . . 6

2.5.2 Adding a New *Other parameters* Menu Choice . . . 7

2.5.3 Adding a New Screen . . . 7

2.6 Suggestions . . . 7

3 Simulator **8**

3.1 Makefile . . . 8

3.2 Files . . . 10

3.3 Data Structures . . . 12

3.4 Notes . . . 13

3.4.1 Output . . . 13

3.4.2 Periodic Tasks . . . 13

3.4.3 Functions . . . 14

3.4.4 Miscellaneous . . . 18

3.5 Modifications . . . 18

3.5.1 Adding a New Function . . . 18

3.5.2 Adding a New Function Type . . . 19

3.5.3 Defining a New Event . . . 20

3.5.4 Modifying a Data Structure . . . 20

3.6 Suggestions . . . 20

4 Animated Run-Time Display **21**

4.1 Makefile 21

4.2 Files 22

4.3 Data Structures 22

4.4 Notes 23

4.5 Modifications 23

4.5.1 Adding a New Event 23

4.6 Suggestions 23

1 Overview

This document is in the directory:

`freal:~gene/testbed/doc/prog/`

There is a Makefile to create it.

The Spring Simulation Testbed is divided into three modules:

1. Configuration Interface
2. Simulator
3. Animated Run-Time Display

This document describes in detail how to modify all three modules. Please note that the Testbed was designed in a modular manner to make continuing modification a pleasant process. There are several guidelines to follow:

- The data structures have access routines to set and get the values. You must use these, and not access the structures directly.
- All output and statistics are produced by reporting events to the routine *io_event*. Add new events as necessary, but continue to go through this routine so there is a single place to modify.

I cannot think of others at the moment, but I'm sure there are more.

2 Configuration Interface

The Configuration Interface is written in C, using SX, a package built on top of X-windows.

2.1 Makefile

This module is combined with the Simulator. See the Simulator section for a description of the makefile.

2.2 Files

This module is in the file:

`min:~gene/testbed/src/iface-routines.c`

and has its own header file:

`min:~gene/testbed/inc/idef.h`

It also uses the Simulator header files:

`min:~gene/testbed/inc/param-ext.h`
`min:~gene/testbed/inc/macros.h`

Param-ext.h contains all the external definitions for all of the parameters set in the interface. It and *param.h* must be kept consistent.

2.3 Data Structures

Each screen has an entry in the *windows* array, consisting of the window id, flag to determine if it has been created, and title. Each entry in the *WinMenu* array maps to one of the screens in *windows*.

Each entry has a mnemonic of the form *I_XXX*. The array *entries* contains *ENTRYREC* records of all the information needed about an entry:

- Prompt to request data.

- Pointer to *MENUREC* record containing information about an associated menu, if any (set up through the array *entryMenus*):
  - Index back to the *entries* array.
  - Address of array of menu choice strings.
  - Number of menu choices.
  - Window that contains the menu.
- *VARREC* record containing information about the variable:
  - Address of variable that contains the default value for this item. (1 - see *Notes*)
  - Type of variable.
  - Address of validation routine.
  - Some bounds checking information, if relevant.
- An array of *EWINREC* records, one for each sub-entry in this group (for example, one for each node):
  - Address of associated variable. (2 - see *Notes*)
  - Index back to the *entries* array.
  - Index back to the *EWin* array for this entry.
  - Physical space for the user's response.
  - Window tht contains the entry.
  - Handler for the entry.

2.4 Notes

Defaults are currently set up through the file *param.h* It would be better if there were a default configuration file that was read if the module were run without the *-i* or *-r* command line options.

There are a couple of tricky parts. First, the entry mnemonics *I_FUNCCHOICE_XXX* are set up in the same order as the *F_XXX* mnemonics in *def.h*. *I_FUNCCHOICE* is defined to be the same as the first of them. This way, they can be looped through in order, and the *I_XXX* and *F_XXX* mnemonics for the same funtion type can be found easily.

The second tricky part is what happens in *Initialize* after the comment *initialize the variables from the defaults.* Each variable to be set in the interface corresponds to one *ENTRYREC* record. This variable will have as many entry windows as its dimensionality. For example, *SimLength* has one entry window, while *AllowSig* has *NumNode * NumSig* entry windows. For each entry window, there is one *EWINREC* record, stored in the *EWin* array in the *ENTRYREC* record. The routine *InitEWRec* initializes the *EWin* records.

The *Addr* field of the *EWINREC* record ((2) above) is what causes the complexity. It must be set to point to the exact location where user input into the corresponding entry window should be written. For a 0-dimensional variable, that is the address of the variable itself. For arrays, it is the address of the array slot. Also, defaults for the input variables are set up in *param.h*. For 0-dimensional variables, like *SimLength,* there is just a single value. For 1-dimensional variables, like *NumProc,* the zeroth array slot is initialized, and the rest of the array must be copied from that. For 2-dimensional arrays, such as *AllowSig,* the zeroth row is initialized, and the remaining rows must be copied from that. The place it should be copied to is what is pointed to by the *Addr* field. If defaults were set up from a file, this part could be simplified.

When the *entries* array is initialized (see the large *static ENTRYREC entries[NUM_ENTRIES]=* statement), the *Default* field in the *VARREC* record ((1) above) is initialized. The *Addr* reference and value are set up relative to this. The *addrOffset* passed to *InitEWRec* is the difference between the *Default* and the desired *Addr*. The *valueOffset* passed to *InitEWRec* is the difference between the *Default* and where the default for this variable lies. For 0- and 1-dimensional variables, this will be zero, since the default is copied from either the variable itself or the zeroth array

slot. For 2-dimensional arrays, it is an offset into the zeroth row of the array, so the row can be copied through the rest of the array.

The third thing that will not be immediately apparent is how SX handlers for entries are managed. When an entry is created, a handler is created to be invoked on any key press. When this handler is invoked, it replaces itself with a handler to be invoked upon leaving the entry window. When this handler is invoked, it puts back the original handler, reads the response, validates it, sets the corresponding variable, and writes back to the screen.

In addition, there are a few hard-coded lines in *iface-routines.c* that disable some function choices. In the routine *MapMenu,* look for the case statement that sets the *MenuMask.* I'm using an octal representation of a bit vector, where there are 0s for disabled entries. The first entry on the menu corresponds to the lowest bit in the mask. You'll have to change these if you add/change the functions.

2.5 Modifications

2.5.1 Adding a New Entry

- Add variable(s) to record the user input in *param.h* and *param-ext.h* if necessary.
- Add code to read and write the new variable(s), if any, to *io_read, io_write,* and *io_report.*
- Add code to *io_set* as necessary.
- Increment *NUM_ENTRIES.*
- Add a new *I_XXX.* Note: the *I_FUNCCHOICE_XXX* ones have to be contiguous.
- If it has a menu, increment *M_NUM* and add an entry to *entryMenus.*
- Add an entry to *entries* in the correct place. (They are indexed by *I_XXX*).

- Initialize it in *Initialize*.
- Display it in *MapEntries* on the desired screen.

2.5.2 Adding a New *Other parameters* Menu Choice

- Increment *W_NUMMENU*.
- Add a new entry to *WinMenu*.

2.5.3 Adding a New Screen

- Increment *W_NUM*.
- Define a new mnemonic *W_XXX*.
- Add a new entry to *windows*.

2.6 Suggestions

Just a few, there's probably lots to do . . .

- Make the interface more robust:
  - If the user selects number of nodes = 1:
    - Disallow signals for global scheduler and bidder
    - Make the selections for the global scheduler and bidder *none*, and the periodicity 0.
  - If the user selects no poller, turn all signalling stuff off.
- Add a screen to enable/disable reporting of certain events.

3 Simulator

The Simulator is written partially in the simulation language SIMSCRIPT (12%) and partially in C (88%). The mechanics of the simulation are in SIMSCRIPT. The modules that exist in the Spring kernel are in C, to facilitate replacement by other researchers. The data structures are also in C, and there is a layer of access routines on top of them, so they can be accessed uniformly from both languages. See the section on Data Structures for this module for more detail. The constants and mnemonics that are needed by routines in both languages are in *def.h* and *preamble.sim.* See the description of *make defs* below.

C routines can be called from SIMSCRIPT if they are defined in the *preamble.sim* as C routines. They are called just as they are from C (but with the SIMSCRIPT *call* statement), but the arguments lists are limited. SIMSCRIPT routines can be called from C by putting the routine name in capital letters and prefacing with an *R.* For example, there is a SIMSCRIPT routine *get_time,* which would be called from SIMSCRIPT as *var = get_time,* but from C as *var = RGET_TIME().* I know, it's a hack. I'm not sure how I figured it out. Also, you can pass character strings in one direction, but not the other. I forgot which. One more thing, I think I had problems calling mixed-case C routine names from SIMSCRIPT.

3.1 Makefile

The makefile is:

`min:~gene/testbed/src/Makefile`

Options:

- **Make -** Will recompile and link everything necessary to create a new executable. It assumes that all *.c* files depend on the header file *structs.h*, which in turn

depends on *def.h.* It assumes that all *.sim* files depend on *preamble.sim.* In addition, there are individual dependencies set up for other header files, modify this list as necessary.

- **Make defs** - *Preamble.sim* contains all the definitions in *def.h,* plus many more. Whenever *def.h* is modified, do a *make defs,* and then replace the section of *preamble.sim* between the arrows with the contents of the created file *defs.* (This is documented in both *preamble.sim* and *def.h.)*
- **Make depend** - Creates the last section of the Makefile. Whenever source files are created or destroyed, do a *make depend* to update the Makefile.
- **Make clean** - Gets rid of some trash. (not *.o* files.)
- **Make back** - Copies all source and header files to a backup directory and compresses them.
- **Make lint** - Does a *lint* on all the C source and header files into the file *lint.out.*

One note - the SIMSCRIPT compiler works by translating the file into a C version (with a *.c* extension) and then compiling it.

1. This is why compiling SIMSCRIPT files takes so long.
2. Don't have two files with the same root name, one with a *.c* extension, and one with a *.sim* extension.
3. If the compile fails due do errors, the *.c* version may be left around. Delete it, so it doesn't look like regular C source code to the Makefile.

It would be great if you changed this makefile so that if any of the files shared with the Animated Run-Time Display are changed, they are copied to that machine. I had a problem with rcp-ing in that direction, I could only do it from freal.

3.2 Files

The Simulator has source files in:

`min:~gene/testbed/src/`

and header files in:

`min:~gene/testbed/inc/`

A quick summary of files:

HEADER FILES:

- **def.h** - basic, system-wide definitions.
- **macros.h** - a few macros.
- **param-def.h** - definitions needed by *param.h* and *param-ext.h.*
- **param-ext.h** - external definition of parameters set by interface.
- **param.h** - parameters set by interface.
- **print_dsp.h** - stuff needed by L. Molesky's *print_dsp.c.*
- **reader.h** - definitions needed by workload reader.
- **structs.h** - basic, system-wide data structure definitions.

SIMSCRIPT SOURCE:

- **debug.sim** - some SIMSCRIPT debugging aids.
- **dispatcher.sim** - the dispatcher and dispatcher wake-up routines.
- **init.sim** - SIMSCRIPT initialization routines.

- **main.sim** - main SIMSCRIPT routine.
- **misc.sim** - miscellaneous SIMSCRIPT routines.
- **preamble.sim** - SIMSCRIPT global definitions, etc.
- **processor.sim** - the processor and processor interrupt routines.
- **snapshot.sim** - the snapshot event generator.
- **stopsim.sim** - the end-the-simulation process.
- **subnet.sim** - all the network stuff and most of the global scheduling routines.
- **taskgen.sim** - the task generators and the process that runs the poller.

C SOURCE:

- **cutoff.c** - routines called by the scheduler to manipulate the STT.
- **disp-routines.c** - routines for the dispatcher ADT.
- **dv-routines.c** - the routines that are needed on freal AND min.
- **func-routines.c** - routines for the function ADT.
- **gsched.c** - the global scheduler.
- **guar-routines.c** - guarantee routines.
- **io-routines.c** - routines for i/o of parameters and the event handler. NOTE: all output goes through the routine *io_event.*
- **misc-routines.c** - miscellaneous C routines.
- **node-routines.c** - routines for the node ADT.
- **poller.c** - the poller.

- **print_dsp.c** - routine (from L. Molesky) to print the STT.
- **proc-routines.c** - routines for the processor ADT.
- **q-routines.c** - routines for the queue ADT.
- **res-routines.c** - routines for the resource ADT.
- **sched-routines.c** - local scheduler routines.
- **sig-routines.c** - routines for the signal ADT.
- **task-routines.c** - routines for the task ADT.
- **tsinput.c** - routine to read the workload.

3.3 Data Structures

I will present a summary here, see the Design Document, section *Low Level Design - Abstract Data Types*, for more detail.

- **CODEVECT** - in structs.h. Array of pointers to functions. See the Design Document, section *Simulate Task Execution*, for more detail.
- **TIMEVECT** - in structs.h. Array of times. See the Design Document, section *Simulate Task Execution*, for more detail.
- **RESVECT** - in structs.h. Array of resource usage.
- **SORTVECT** - in structs.h. Array of sort keys.
- **TASKT** - in structs.h. Structure for task information.
- **QUEUET** - in structs.h. Structure of queue information. Queues have linked lists of tasks.

- **DISPT** - in structs.h. Structure of dispatcher information. Contains a pointer to the queue header for its STT, as well as pointers to the first and last tasks in its dispatching chain.
- **PROCT** - in structs.h. Structure of processor information. Contains a pointer to the record for its dispatcher.
- **SIGT** - in structs.h. Structure of signal information.
- **NODET** - in noderoutines.c. Structure of node information. Contains pointers to processor, queue, and signal structures.
- **Function data structures** - in func_routines.c. Structures for replaceable function information.

3.4 Notes

There is much more detail in the Design Document, but I will mention some of the more important things here.

3.4.1 Output

All output is event-driven, and is produced by calls to the routine *io_event* in *io-routines.c*. For verification, some standard output should be produced. Just add code to this routine for the appropriate events, and send the verification output to a file. The routine *print_dsp_sim* is called after each successful guarantee. This can be used as part of the verification. Note that it has been slightly modified from the original.

3.4.2 Periodic Tasks

Periodic tasks are not scheduled. What I have now is a hack to get them to work. Each periodic task type has its own task generator, and they are generated and placed

into the STT directly. This should be changed.

What I would do is create a queue of currently active periodic task types. Each periodic task has an arrival and a departure between which it resides on the aforementioned queue. The local scheduler, for example, would arrive at time zero and never depart. The scheduler would look at this queue to create a schedule of tasks to run, and place copies of the tasks in the STT, deleting the tasks from the queue only at/after departure time. The queue could be the same as the Candidate Queue, as long as the scheduler(s) know to handle periodic and non-periodic tasks differently, especially the fact that non-periodic tasks are removed when scheduled and periodic tasks are not. I would just try to have the dispatching be uniform with the way non-periodic tasks are dispatched, and allow the scheduling to be replaceable.

3.4.3 Functions

The way replaceable functions are handled is much simpler than it seems at first glance. The relevant code is in *func_routines.c* and *param.h*.

For each function type the following data structures (from *param.h*) are used:

Name of the function type:

```
char *FuncTypeName[NUM_FTYPE]={
    "Reclaiming",
    "Local Scheduling",
    "Guarantee",
    "Heuristic Function",
    "Global Scheduling",
    "Signal Selection",
    "Signal Request Honoring",
    "Task Resumption After Interrupt",
    "Polling",
```

```
    "Interrupt Handler",
    "Bidding"};
```

Number of functions currently defined for the type:

```
int NumFunc[NUM_FTYPE]={3, 2, 5, 4, 6, 2, 2, 3, 2, 2, 2};
```

Name of each function defined for each type:

```
char *FuncName[NUM_FTYPE][MAX_FUNC]={
    {"Start Task Immediately", "Start Task at SST",
       "Start Task When Has Earliest DL & Resources Available"},
    {"Earliest Deadline First", "Heuristic"},
    {"None", "Plain", "O(n)", "Backtracking", "O(n) & Backtracking"},
    {"None", "Min Deadline", "Min (Deadline + W * SST)", "Min Laxity"},
    {"None","Best Node Bidding","Probabilistic Bidding","Focussed Addressing",
       "Combined","Random"},
    {"None", "FCFS"},
    {"None", "Always Honor Request"},
    {"None", "Always Resume Task", "Never Resume Task"},
    {"None", "Poller"},
    {"sig_handle", "sig_set_proc_intr"},
    {"None", "Bidder"}
};
```

Number of code sections used for each function for each type: See the Design Document, section *Simulate Task Execution*, for more detail on code sections.

```
int NumCode[NUM_FTYPE][MAX_FUNC]={
    {1, 1, 1},
```

```
    {1, 1},
    {1, 1, 1, 1, 1},
    {1, 1, 1, 1},
    {1, 1, 1, 1, 1, 1},
    {1, 1},
    {1, 1},
    {1, 1, 1},
    {1, 1},
    {1, 1},
    {1, 1},
};
```

Names of routines to be called for each code section for each function for each type (note that each function has only one code section):

```
CODEVECT CodePtr[NUM_FTYPE][MAX_FUNC]={
    {{task_get_arrival}, {task_get_sst}, {sched_starttime_dlfirst}},
    {{sched_dl_first}, {sched_heur}},
    {{NULL}, {guar_nolin_noback}, {guar_lin_noback}, {guar_nolin_back},
       {guar_lin_back}},
    {{NULL}, {task_get_dl}, {hf_dl_and_sst}, {task_get_laxity}},
    {{NULL}, {gsched}, {gsched}, {gsched}, {gsched}, {gsched}},
    {{NULL}, {sig_pik_fcfs}},
    {{NULL}, {return_true}},
    {{NULL}, {return_true}, {return_false}},
    {{NULL}, {Poller}},
    {{sig_handle}, {sig_set_proc_intr}},
    {{NULL}, {bidder}},
};
```

The function ADT routines and the interface routines are driven by these data

structures. At system configuration, a selection is made of one function for each function type (except interrupt handler, that is done internally). Internal (to *func-routines.c*) data structures are set to indicate which selection was made. A task structure is initialized for each function type with code pointers set to those in *CodePtr[type][selection].*

A replaceable function is run by calling *func_exec_one* with the node id of interest, function type, and a single parameter to pass. This parameter should be cast to *char *,* and cast back to the desired type in the function itself. If more than one parameter is needed, create a structure with all the necessary information and pass a pointer to it. It assumes that the function returns an integer. The currently defined functions of each type assume a certain interface (a blank entry means no particular assumptions):

TYPE	INPUT	OUTPUT	SIDE EFFECTS	AVAIL. GLOBALS
F_RECL	task pointer	task start time		
F_LSCHED	Node id	ERROR/NOERROR	trans. tasks from Cand Q to STT	
F_GUAR	SchedQ	TRUE/FALSE (G/NG)		Max tasks/h-func./ backtrack
F_HF	task pointer	heuristic value		weight
F_GSCHED	Node id	ERROR/NOERROR	send out task	
F_SIGPIK	Node id	signal index		signal vectors
F_SIGOK	Node id	TRUE/FALSE		signal vectors
F_SIGRT	Node id	TRUE/FALSE		signal vectors
F_POLLER	Node id	NOERROR		signal vectors proc. int. bit
F_IH	Node id	ERROR/NOERROR	handle interrupt	
F_BIDDER	Node id	?	?	

3.4.4 Miscellaneous

USE THE DEFINED DATA STRUCTURE ACCESS ROUTINES TO ACCESS THE DATA STRUCTURES! This is the single most important rule. If you cannot do something with the ones that are there, or some combination, create more, but don't access the structures directly!

Each processor has a dispatcher with the same id. I may use the term processor id/dispatcher id interchangeably.

Tasks may be dispatched to a processor other than the one to which they are assigned if the dispatching option is set to *Independent.* However, they must still have a processor id of one of the processors in the group to which they may be dispatched. This is because the scheduler uses the processor id to figure out which STT to put the task on.

Tasks may be sent to another node for scheduling. However, they will retain their processor ids and resource usage vectors. These must make sense on the new node.

SIMSCRIPT array start at 1, C arrays are zero-based. In order not to have to add/subtract 1 all the time, I made my important C arrays one larger than necessary, and leave the zeroth spot unused. When you see an array dimensioned by *NUMNODE+1*, or almost anything+1, this is the reason. So, be careful with loops that traverse arrays. I broke this rule with the interfaces where I have arrays dimensioned [0 ... n-1].

3.5 Modifications

3.5.1 Adding a New Function

Note: You will not have to modify anything on the Configuration Interface if you follow these steps. It is driven by the header files. You may have to change the hard-coded lines in *iface-routines.c.* Look at the *Notes* section for the Configuration

Interface. Sorry.

- Write your new function using the conventions in the table above. Use the defined ADT routines for data access. Look at some existing code for examples. Look at the correspondence between Lory's guarantee routines and mine for a good idea of how to use the ADTs. Also, see the section on *Data Structures*, and the Design Document, section *Low Level Design - Abstract Data Types*, for more detail on the ADTs.
- Increment the correct slot in *NumFunc* in *param.h.*
- Add a new entry to the corresponding slot in *FuncName, NumCode,* and *CodePtr* in *param.h.*
- Change the default in the corresponding slot in *FuncChoice* in *param.h* if desired.

3.5.2 Adding a New Function Type

- The basics:
  - Increment *NUM_FTYPE* in *def.h.*
  - Define a *F_XXX* mnemonic for it in *def.h.*
  - Add an entry to *FuncTypeName, NumFunc, FuncName, NumCode,* and *CodePtr* in *param.h.*
  - Add defaults for node 0 to *FuncChoice, FuncPer, FuncWC, FuncAC,* and *FuncPre* in *param.h.*
  - Add functions for this new type as in the previous section.
- For the interface (note the similarity to *Adding a New Entry* in the Configuration Interface section):
  - Increment *NUM_ENTRIES* in *idef.h.*

- Increment *M_NUM* in *idef.h* (since it will have a menu).
- Define a *I_FUNCCHOICE_XXX* mnemonic for it in the correct order in *idef.h*.
- Add an entry to the correct spot in *entries* in *iface-routines.c*.
- Add an entry to *entryMenus* in *iface-routines.c*.
- Map it to the screen somewhere in *iface-routines.c*.

3.5.3 Defining a New Event

Routines all over the testbed generate events that are reported to *io_event* in io-routines.c. I have enabled only a few of them. See the section on the Run-Time Display for details on what to do with generated events.

- Define a *E_XXX* mnemonic in *def.h*.
- Have the event be reported somewhere.
- Modify the array *str* in *io_event*.
- Add a case to handle the event in *io_event*.
- Add a case to handle the event in *UpdateData* in *driver.c* in the Run-Time Display.

3.5.4 Modifying a Data Structure

In addition to modifying the structure, add access routines to set and get the new field. Look at the initialization routine for the ADT.

3.6 Suggestions

- Do periodic tasks!

- Add more functions.
- Add a way to dump out something that can be used by Lory's verification tool.
- Merge the four guarantee routines into two, or even one, where different algorithms are simulated by the choices made for *Additional Scheduling Parameters.* To do so, follow the steps for *Adding a New Function,* but delete one instead.

4 Animated Run-Time Display

The Animated Run-Time Display is written in C, using Dataviews, a commercial package for animated display. The views were built using the DVdraw utility, and the animation is produced through calls to DVtools routines.

4.1 Makefile

The makefile is:

`freal:~gene/testbed/src/Makefile`

Options:

- **Make rcp -** Copies all the necessary files from min. It does not check if they have been modified since last copy. Do this if you even suspect the needed files have been modified on min.
- **Make -** Will recompile and link everything necessary to create a new executable. It assumes that all *.c* files depend on the header file *def.h.* In addition, there are individual dependencies set up for other header files, modify this list as necessary.
- **Make depend -** Creates the last section of the Makefile. Whenever source files are created or destroyed, do a *make depend* to update the Makefile.

- **Make clean** - Gets rid of some trash. (not *.o* files.)
- **Make lint** - Does a *lint* on all the C source and header files into the file *lint.out.*

4.2 Files

There are two source files that are solely for the Run-Time Display:

`freal:~gene/testbed/src/driver.c`
`freal:~gene/testbed/src/dummy.c`

This module also shares a source file and several header files with the other modules. The shared source code file is:

`freal:~gene/testbed/src/dv-routines.c`

The header files are in:

`freal:~gene/testbed/inc/`

If ANY of the shared files are changed on min, you MUST do a *make rcp* for this module!

Driver.c contains everything important to produce the animation. The definitions in *dummy.c* are there so *param.h* can be loaded.

4.3 Data Structures

The only major data structure is *ViewRec*, an array of records of type *VIEWT* containing, for each view:

- File name of view
- Drawport for view
- Drawing for view

4.4 Notes

The only tricky part is that for each variable in each view, I define a *xxxName* variable, and a *xxxPtr* variable. The name is the same as the name of the data source variable, and I use it to assign the pointer variable to the address of the buffer that Dataviews uses for the data source. Then, I write my data to this pointer.

Right now, I have only a few active events:

- E_TASK_ARRIVE - task arrives externally, I make the connector to the node flash green.
- E_SUB_TASK_RECV - task arrives from another node, I make the connector to the node flash blue.
- E_TASK_SEND_YES - task is sent out, I make the connector to the node flash red.
- E_SNAPSHOT - I update the guarantee ratio and utilization information.

4.5 Modifications

4.5.1 Adding a New Event

- See the section *Defining a New Event* above.
- Use DVdraw to add new items to the display and new variables as necessary.
- Define a *xxxName* and *xxxPtr* for the variable(s) as necessary.
- Add code to *UpdateData* to update the new variables.

4.6 Suggestions

- Have the module be run from a script file that sets up the Dataviews environment and then passes the command line options to the program.

- Modify *UpdateData* so it skips a line when it sees an event it doesn't recognize. That way, if there is one, it won't blow up. There is an untested routine *SkipLine* in *dv-routines.c*.

- Rewrite the data source variables as function data sources that know where to get their own data.

- Do something to center the nodes/processors if there are fewer than five. Also, find a better way to make them invisible!

- Add a third view, for processors, that comes up when you click on a processor. It could show the details of tasks that have run, are running, will run, etc.

- Add a view for resource usage, with some sort of graphic display to show utilization.

- Add a way to single step through the simulation.

- Add a way to control the time - with some correspondence to real time.

- Have the bus do something when there are tasks on it. I would set up color thresholds to represent levels of busyness.