# Marvel
# Implementors Guide

Michael H. Sokolsky[1]
Naser S. Barghouti[2]
Columbia University

Technical Report CUCS–428–89

March 8, 1990



[1]Supported in part by the Center for Advanced Technology.
[2]Supported by the Center for Telecommunications Research.

## Abstract

This document is an implementors manual for the MARVEL software developement environment. It discusses the technical details of the system, rather than providing a description of the system. It is intended for those people doing actual development work on MARVEL. References are provides throughout the manual for background information.

# Contents

# Chapter 1

# Introduction

This manual is an *implementors* guide to MARVEL 2.5. As such, it does not make any attempt to teach the reader about MARVEL, either theoretically or practically. It is highly recommended that readers first have a useable knowledge of the system before starting on this guide. A theoretical background for MARVEL can be gleaned from [SK89, GEKS90, GEKS88, GEKP88, BK88, FK87] and [KF87], while a practical working knowledge of MARVEL can be gained by consulting [CSB89] and by banging on the system.

We begin by presenting all the data structures the system uses. This chapter should not be dwelt upon first. We then present the command interpreter, including the line oriented interface. We then present the X interface. Those not dealing with the X interface can skip this chapter without worry of missing anything. We continue with the Marvel built in commands in chapters 7 and 8. Chapters 9 and 10 discuss the heart of MARVEL, the evaluation of rules, and the opportunistic procssing of rule chains. Chapter 11 discusses the various rule overloading facilities in MARVEL Chapter 12 discusses the Marvelizer, the facility used to import existing code into MARVEL. This module is very independent of the rest of the system, and can be skipped unless it is to be worked on. The following three chapters discuss the process of loading and saving data models and objectbases. This includes the parsing of MSL (*Marvel Strategy Language*) strategies. Chapter 17 discusses some functional that should probably be put elsewhere, but have not for some reason. Finally, we conclude with a chapter on querying the objectbase. and data and process models currently loaded.

# Chapter 2

# General Information

This Chapter discusses some general information about the structure of MARVEL , and modifying code. It has been put in the begining of this manual so someone might see it.

## 2.1 Description of Directories

At Columbia, MARVEL lives in `/proj/marvel` (not really, but that is what the network tells you). This is the "Release" area, which simply means that it is the current state of the system, not a work area. We have a two stage work/release environment, rather then a work/local release/global release strategy.

All shared code lives under the shared directory. That directory contains a directory for each logical module of code within Marvel. This is master source, and should stay read only, according to RCS. When working with source, it should be checked out elsewhere (see below). All source code is RCS'd, but Makefiles are NOT, as they are auto generated (see below). This code is compiled into a ar library called marvel.a. Make should NOT be run directly in these directories, as the Makefiles need special things that scripts described below set up.

All program specific code lives in a programs directory. This directory contains loader and marvel, which are the two programs that comprise the executable part of the system. Under these directories are subdirectories that modularize the programs. A library is created for each program, called loader.a and marvel.a, respectively. As with the shared directory, Makefiles are auto generated, and make should not be directly used on these modules.

Include files live in a directory called "include". All definitions go here. There is a global include file called marvel.h for anything that the entire project uses. All include files are RCS'd.

There is a directory called lib where the released code libraries. lives. The presence of this library makes it VERY easy for each individual to test her or his own changes to the code. Libraries for the appropriate architecture you are using are found under the appropriate subdirectory. Makefile templates, and File and function header templates also live here. They are all RCS'd. Tags files, both for vi (tags) and emacs (TAGS) are also kept here. They are generated, thus not RCS'd.

There is a directory called bin for related binaries and scripts. Actual binaries are found in appropriate subdirectories, depending upon the hardware. There are scripts in the bin directory that figure out the kind of hardware, and execute the appropriate program.

The code is currently compiled and tested using sun3 and sun4 and IBM RTs. The sun's run Sun OS 4.01, and the RT's AIX. The scripts in bin are all ksh scripts, but should work in a csh environment. The reserve script uses a small C program called getrcsname, it is compiled and it's src lives in bin/src (under RCS, of course).

There is a flatsrc directory (which gets remade with every makep) for use with the debugger. It contains all the .c and .h files in src.

There is a directory called help where all the help files live. These are saved with RCS. If you make modifications to a Marvel command, the appropriate help file should be updated.

The examples directory contains all the various Marvel environments available. The only one tested with the final release of the system is cmarvel. The others were written by project students, and have not necessarily been maintained since the projects were completed. The MSL language has been updated, so these environments might not completely work now.

## 2.2 Creating the Marvel Release

In the bin directory, there are several scripts which you might want to know about. All these scripts require three important things, in fact these are ABSOLUTELY crucial to marvel's running:

1. A program called arch somewhere on the users search path. It should return:

   sun3 - for sun3 machines
   sun4 - for sun4 machines
   ibmrt - for ibm RT's
   mips - for dec RISK machines (3100's and 2100's)

2. An environment variable called PROJECT, set to the root marvel directory. In our environment, it is set to /proj/marvel, however this is installation dependent.

3. Addition of the MARVEL 's bin directory somewhere near the begining of your search path.

## 2.3 Scripts

To regenerate Makefiles for the entire system run `makemf`. This uses a binary called `mkmf`, which you will find along with the release. Output is put in a logfile called `makemf.log` in the PROJECT directory. This script only needs to be run if a file or module or program is added or removed from the system. The script automatically figures out what is there, it should not need modification upon such addition.

To create a binary for all of marvel, there is a script called `makep`. It does not take any arguments. It created the shared.a, marvel.a and loader.a libraries, and then binaries for the machine it is working on. Starting from scratch on a sun 4/60 this takes 15-20 minutes. Output is put in a logfile called `makep.log` in the PROJECT directory. Makep also recreates the tags databases, and the flatsrc directory. In addition, `makep` reruns `makemf` if this is the first time in a row you have compiled on a particular hardware.

There are two scripts called `deposit` and `reserve` which are simple, friendly front ends to `co` and `ci` for checking files in and out. They will put the checked out file where the user desires it (preferably some work area in the user's own space). You can only reserve one file at a time (deposit works on multiple files), but the scripts are otherwise robust. You can also use co -l and ci -u. Note that the RCS commands do not come with MARVEL .

## 2.4 Making Program Modifications

I recommend a procedure similar to the following for working on the code:

1. make a work directory somewhere in your home area.
2. checkout files as needed into this work area.
3. create a Makefile. A suggested template is in the lib subdirectory, called Makefile.local.
4. modify the makefile to include the c files you have checked out. I use mkmf to do this step.
5. BIG STEP - modify the code.
6. compile by running make.
7. run the program you have created on any test data you desire.
8. repeat the last 3 steps, checking out more files and adding them to the makefile as needed.
9. When you are finished, check in the files to the release source area. Note that you should ONLY check in files which compile and lint. This way, the release source will be assured to work.
10. At this point makep can be run to create a new master marvel executable with your changes in it. Just run makep.

If you understand the above, the following will be obvious.

- use reserve to check out
- use deposit to check in

- use make to compile local programs, and only use makep for the release source.
- Once the first several steps are set up above, only the makefile needs editing upon checkin and out of code.
- Make use of tags.
- Note that you are assured of getting an entire marvel because of the code living in a library, and your object files being loaded before the library.

# Chapter 3

# Data Structures

This chapter discusses all the various data structures used in Marvel. It will probably not be completely comprehensible without the context of particular subroutines that use the datastructures described within, so it should probably should only be skimmed, and then used a reference.

We try to break the presentation down somewhat analogously to the remainder of the chapters. We do not present data structures in each chapter, because some of our modules do more sharing of them then would be appropriate for such a presentation.

## 3.1 Interface

## 3.2 Evaluator and Opportunist

## 3.3 Objectbase Manager

## 3.4 Links

## 3.5 System Messages

## 3.6 Creating Data Structures

There are system routines to create and initialize all data structures in Marvel. malloc() or some similar allocation routine should **NEVER** be used alone. The front end to all these routines is `make_struct()`, below.

### 3.6.1 make_struct()

```
physadr make_struct(type, name)
int type;
```

```
char *name;
```

This routine is intended to turn allocating of marvel structures more object oriented, in the sense of hiding details. NEVER use malloc to allocate and initialize structures, make_struct() does it all for you. ALWAYS use make_struct(), to avoid maximal debugging headaches.

type is defined in the appropriate include file, and is an integer which represents the structure in question.
name is the name of the primary name field of the record in question. For records which do not have such a field, just supply NULL. Name is copied, so the memory need not be persistent.

make_struct() returns a physadr, which is a guaranteed byte aligned pointer. The results should be cast in some appropriate fashion. This should ease some ofthe problems that might be encountered when going to a binary objectbase format.

In general, all pointers other then the main name (or similar field) are set to NULL. Numbers are usually set to -1, but that varies upon the structure.

These routines make one of something, for example, make_struct with a QUEUE_S flag passed will make an entry of a queue, RATHER THAN an entire queue.

### 3.6.2 make_act_args()

```
physadr make_act_args(name)
char *name;
```

Make an act_args. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.3 make_actlist()

```
/*ARGSUSED*/
physadr make_actlist(name)
char *name;
```

Make an actlist. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.4 make_attribute()

```
physadr make_attribute(name)
char *name;
```

Make an attribute. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.5 make_binding()

```
/*ARGSUSED*/
physadr make_binding(name)
char *name;
```

Make a binding. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.6 make_chains()

```
/*ARGSUSED*/
physadr make_chains(name)
char *name;
```

Make a chains. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.7 make_class()

```
physadr make_class(name)
char *name;
```

Make a class. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.8 make_class_q()

```
/*ARGSUSED*/
physadr make_class_q(name)
char *name;
```

Make a class_q. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.9 make_cmd_line()

```
/*ARGSUSED*/
physadr make_cmd_line(name)
char *name;
```

Make a cmd_line. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.10 make_cond()

```
/*ARGSUSED */
physadr make_cond(name)
char *name;
```

Make a cond. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.11 make_graphic_info()

```
/*ARGSUSED*/
physadr make_graphic_info(name)
char *name;
```

```
Make a graphic_info.  This is the non generic, low level routine.
The generic routine make_struct() should be used instead.
name is unused here.
```

### 3.6.12 make_info_level()

```
/*ARGSUSED*/
physadr make_info_level(name)
char *name;
```

```
Make an info_level.  This is the non generic, low level routine. The
generic routine make_struct() should be used instead.  Name is unused
here.
```

### 3.6.13 make_inherlist()

```
physadr make_inherlist(name)
char *name;
```

```
Make an inherlist.  This is the non generic, low level routine. The
generic routine make_struct() should be used instead.
```

### 3.6.14 make_instance()

```
physadr make_instance(name)
char *name;
```

```
Make an instance.  This is the non generic, low level routine. The generic
routine make_struct() should be used instead.
```

### 3.6.15 make_ipcdbitem()

```
physadr make_ipcdbitem(name)
char *name;
```

Make an ipcdbitem. This is the non generic, low level routine. The generic routine make_struct() should be used instead.
name is used for the printname field.

### 3.6.16 make_link()

```
/*ARGSUSED*/
physadr make_link(name)
char *name;
```

Make a link. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.17 make_list_node()

```
/*ARGSUSED*/
physadr make_list_node(name)
     char *name;
```

Make a list_node. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.18 make_obj_list()

```
/*ARGSUSED*/
physadr make_obj_list(name)
char *name;
```

Make an obj_list. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.19 make_obj_list_info()

```
/*ARGSUSED*/
physadr make_obj_list_info(name)
char *name;
```

Make an obj_list_info. This is the non generic, low level routine.
The generic routine make_struct() should be used instead.
Name is unused here.

### 3.6.20 make_own_link()

```
/*ARGSUSED*/
physadr make_own_link(name)
char *name;
```

Make a own_link. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.21 make_pre_post()

```
/*ARGSUSED*/
physadr make_pre_post(name)
char *name;
```

Make a pre_post. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.22 make_pred_table()

```
/*ARGSUSED*/
physadr make_pred_table(name)
char *name;
```

Make a pred_table. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.23 make_projectitem()

```
physadr make_projectitem(name)
char *name;
```

Make a projectitem. This is the non generic, low level routine. The generic routine make_struct() should be used instead.
name is used for the printname field.

### 3.6.24 make_q_element()

```
/*ARGSUSED*/
physadr make_q_element(name)
char *name;
```

Make a q_element. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.25 make_queue()

```
/*ARGSUSED*/
physadr make_queue(name)
char *name;
```

Make a queue. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.26 make_rel_table()

```
physadr make_rel_table(name)
char *name;
```

Make a rel_table. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.27 make_rule()

```
physadr make_rule(name)
char *name;
```

Make a rule. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.28 make_rule_chain()

```
/*ARGSUSED*/
physadr make_rule_chain(name)
char *name;
```

Make a rule_chain. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.29 make_stack()

```
/*ARGSUSED*/
physadr make_stack(name)
char *name;
```

Make a stack. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 3.6.30 make_strategy()

```
physadr make_strategy(name)
char *name;
```

Make a strategy. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.31 make_strlist()

```
physadr make_strlist(name)
char *name;
```

Make a strlist. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.32 make_subsuper()

```
physadr make_subsuper(name)
char *name;
```

Make an subsuper. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 3.6.33 make_symbol()

```
physadr make_symbol(name)
char *name;
```

Make a symbol. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

## 3.7 Miscellaneous

# Chapter 4

# The Command Interpreter

The command interpreter in Marvel in found in a module called interpreter.

## 4.1 Functions

These functions are all found in ci.c, except for the last one, which is in execute.c.

### 4.1.1 command_str_seperate()

```
CMD_LINE_PTR command_str_seperate(str)
     char *str;

   str -- the command line

   This routine takes the string entered by the user as a command and
parses it.  It places each token in the command line (tokens are
separated by spaces) into a CMD_LINE structure, and returns linked
list of all the strucutures containing the tokes.  It can be used on
an entire command line, or any arbitrary part of a command line.  The
CMD_LINE linked list is used by almost all the rest of the routines
that handle user commands.
```

### 4.1.2 expand_and_execute_command()

```
int expand_and_execute_command(cmd_line)
     CMD_LINE_PTR cmd_line;


Take a command line, find the command in either the command table
```

or the rule table, and execute it. An entire name need not be given for this one to operate, just a unique identifying name.

### 4.1.3 go_for_it()

```
void go_for_it()
```

This is the main command processor. If the user wants graphic mode, it is started here. Otherwise, the routine just loops endless, getting more user input, and calling appropriate commands.

### 4.1.4 marvel_init()

```
void marvel_init()
```

Do whatever is necessary to initialize marvel. It allocates memory for the global args array, reads in the marvel startup file in the user's home directory, sets the path, etc.

### 4.1.5 print_commands_cmd()

```
int print_commands_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;
```

cmd_line -- the structure that contains the user's command and all of its arguments.

This routine is local and static to the ci module.

### 4.1.6 prompt_cmd()

```
int prompt_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;
```

```
   cmd_line -- the structure that contains the user's command and all
of its arguments.

    This routine is local and static to the ci module.
```

### 4.1.7 read_startup_file()

```
void read_startup_file()


This routine reads the marvel startup file.  It is basically the
same as execute_cmd(), but there is no authentication string required.
Currently, only the home directory is searched for the startup file,
which should be called .marvelrc.
```

### 4.1.8 usage_cmd()

```
int usage_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;

   cmd_line -- the structure that contains the user's command and all
of its arguments.

    This routine is local and static to the ci module.
```

### 4.1.9 usage_opts_command()

```
int usage_opts_command(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

   cmd_line -- the structure that contains the user's command and all
of its arguments.
   opt -- the number of the user command in the menu.

    This routine is local and static to the ci module.
```

### 4.1.10 usage_opts_star()

```
int usage_opts_star(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- the structure that contains the user's command and all of
                its arguments.
    opt -- the number of the user command in the menu.

    This routine is local and static to the ci module.
```

### 4.1.11 execute_cmd()

```
int execute_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;
```

This is the batch facility for Marvel. A specified file is opened, the first several bytes of the file to be executed must contain an appropriate authentication. Then the commands are read one at a time and executed as if this what the line interface. The command works in the graphics interface by temorarily making Marvel think that it is in the line interface.

# Chapter 5

# The Graphical User Interface

### 5.0.12 Starting the MARVEL Graphic User Interface

The user can start up MARVEL 's Graphic Interface by specifying the -w argument on the command line. When the arguments are parsed in the file main.c, the variable XFACE is set to true if -w was specified. This variable is used by many routines in MARVEL to determine which interface to use.

The routine go_for_it() in the file ci.c is then called, starting the main input routines going. This procedure never returns.

Depending on the interface, go_for_it() either calls start_marvel_xface() in the file xinit.c, or begins an infinite loop of reading the commandline, parsing the command, and executing the command. For the purposes of this chapter, MARVEL starts in the routine start_marvel_xface().

### 5.0.13 Connecting to the X server and getting the necessary resources

The first thing that all X programs must do is open up a connection to the X server. This is done using the XOpenDisplay(), with the display name as the argument. If the display name is NULL, XOpenDisplay() looks at the environment variable DISPLAY instead. XOpenDisplay returns a pointer to a Display structure, which is necessary for most X graphics calls. If NULL is returned, a connection could not be opened, so marvel reverts to the command-line interface.

After getting a display structure, MARVEL attempts to load the three necessary fonts, small_font, normal_font, and bold_font. Reasonable default values are defined in the file xface.h. These default values can be changed in the .marvelrc file. Since all three of these fonts are essential, if loading any of the fonts fails (most likely because they don't exist), MARVEL frees any previously loaded fonts, closes the display connection, and returns to the command-line interface.

After the fonts are loaded, MARVEL opens up six separate windows, one being the root of the other five. The MARVEL window called *root* window is created as a child of the screen root window. The other five windows are created as children of the MARVEL root window. This is necessary so that the window manager only manages the MARVEL root

window, and not the other five windows (a window manager manages only the windows which are children of the root).

After each window is created, MARVEL requests the necessary events from each window. The MARVEL root window receives KeyPress events, so that the user can have the cursor anywhere on the MARVEL window and still be able to type. If only the text window requested KeyPress events, then the cursor would have to be on the text window for the user to be able to type. This is the only marvel window which does not request Exposure events, simply because nothing is ever drawn on the MARVEL root window.

The status window, which appears on the top column and contains information such as what command is currently being executed, what the current object is, and what software revision MARVEL is currently at, requests only ExposureEvents. ExposureEvents are sent whenever part or all of the window needs to be redisplayed.

The main display window appears directly underneath the status window. This is where the object hierarchy and the rule graph are displayed. Since the user can click on any object instead of typing in its name, this window must receive ButtonPress events in addition to ExposureEvents.

The text window appears directly underneath the main display window, and is used for text input and output. Since all keypress events are directed to the MARVEL root window, the text window does not need to request KeyPress events. Exposure events, of course, are requested.

TODO: The menu windows use Mike tanenblat's code, which I haven't looked through yet...

After these windows are created, a global Graphic Context (GC) is created. A graphic context contains information regarding how to draw things in the window, such as line thickness, foreground and background colors, font, etc. The graphic context is usually an argument to all X11 functions that draw something in a window. Some reasonable values are set for this GC.

A global pixmap the size of the display window is now allocated. A pixmap is similar to a window in the sense that it could be drawn to, but different since it is indestructable. This pixmap is needed to redraw the display window whenever an exposure event is received by that window. All output that is to be sent to the display screen should either be drawn to both the display screen or draw it exclusively to the pixmap, then use the routine display_entire_pixmap() to copy the contents of the pixmap to the display window. The latter is faster, since only one thing is being drawn instead of two, but the user receives no feedback about what is going on, since nothing changes on the screen until the pixmap is instantaneously copied over.

TODO: After all the X11 resources are allocated, MARVEL calls set_up_rule_table(), which sets up the data structures required for properly maintaining the rule menu. NOTE: Since this is going to change soon, I'm not going to document it right now.

Then, the event processing loop starts by calling the routine do_main_loop(), in the file xevents.c.

The first thing do_main_loop() does is call paint_status(), which updates the status window with the current software level on the left, current object on the right, and any information provided in the middle. This information is usually the command currently being executed. The the text window is cleared, and all internal variables relating to the

text window are reset, by calling the routine init_message() in the file message.c. The display pixmap is then cleared by calling clear_disp_pixmap(). Finally, the object entire hierarchy is displayed by calling the routine disp_obj() (see Displaying the Objectbase, below.)

### 5.0.14 The X Event Loop

Events are sent to the MARVEL program by the X graphics server as external actions occur. For example, whenever the user presses the mouse button, any window underneath the cursor that requested ButtonPress events will receive one. Exposure events are sent whenever a window becomes erased for some reason, usually because the user moves another window off the MARVEL window, or raises the MARVEL window.

#### Exposure

The X11 graphics system requires each application to restore the contents of its windows whenever that information is lost. For example, restoring a window's contents becomes necessary when the user moves a window, thus exposing the window underneath.

In MARVEL , exposure events are handled different depending on which window needs re-exposing.

To redisplay the *display* window, a *pixmap* is maintained with the current contents of the window. A pixmap is an X11 drawable that, unlike a regular window, cannot lose its contents. Pixmaps, however, cannot be displayed. Before a pixmap can be displayed, however, its contents must first be copied to a window. Drawing to a pixmap is similar to drawing on a window, except that the results would not be immediately visible. When re-exposure of the display window is necessary, the proper portions necessary to complete the display (as dictated by the exposure event) are copied over to the display window.

Although a pixmap is expensive memory-wise, its use is justifiable for two reasons. First, recalculating and redrawing the display line by line is time-consuming. In the worst case, recalculation of the necessary information requires requires two depth-first traversals of the objectbase. In the best case, a single traversal is necessary. For a small objectbase, using a pixmap would be wasteful, since the time required to recalculate and draw everything would be small, but for normal sized objectbases, the time required to recalculate and redraw the display becomes unbearable. Redrawing the "rule dependencies" graph (select print, then graph), is also time consuming, even for a small number of rules.

Also, the pixmap provides additional flexibility, since the display window can be redrawn without knowing what was previously displayed. Without the pixmap, MARVEL would have to remember what is being displayed and how to redraw it.

The menu and the rules windows are completely redrawn when an exposure event is received. Since only a relatively small number of options is displayed at any give time, redrawing the entire window is faster than calculating which parts were erased and attempting redrawing them. Since the contents of the menu window cannot change, redrawing it requires no extra information. The contents of the rule window, however, are constantly changing as different strategies are loaded and unloaded, so a table containing

a list of the currently available rules is maintained. When erased, the rule window is redrawn using the information in this table.

Currently, there are no provisions for redrawing the contents of the options menu window. The options menu window lies underneath the commands menu window, and is used when a command needs additional options.

The text window does not currently support exposure events. The entire text window is simply erased every time it receives an exposure event.

The Status window is completely redrawn every time an exposure event is received.

### Button Press

Whenever the user presses any of the mouse buttons anywhere on the display, a Button Press event is generated and sent to MARVEL . MARVEL acts differently depending on the state when the button was pressed. For example, when marvel is waiting for the user to pick a command, if the user presses the right button in the display window, information regarding the instance selected is displayed in the text window. If the left button is pressed, the current object changes to the instance selected instead. However, while in the browser, the left and right button pan the display in the respective direction, while the middle button zooms the display to the selected instance.

### Key Press

Whenever the user presses a key on the keyboard while the mouse cursor is over the MARVEL window, a Key Press event is generated, and sent to MARVEL . Key Press events are ignored unless the user is prompted for input.

As an input prompt, a black rectangular cursor is drawn in the text window, after the input prompt. The user can now type any alphanumerical character. It will be echoed on the screen. To correct errors, the user can press backspace, causing the cursor to destructively move back one character space. By pressing Control-X, the input buffer is cleared, leaving the cursor immediately after the input prompt, as if starting from scratch. This is equivalent to pressing backspace enough times to move the cursor to the beginning. If Control-C is pressed, the input is canceled. This is equivent of not inputing anything.

A nondestructive backspace key is not implemented.

The mouse buttons are also active while in input mode. Pressing the left mouse button on the main display screen while prompted for input causes the name of the selected instance to be entered into the input buffer. This way, the user can use the names of instances or rules as arguments for commands without having to retype their names. Note that this is equivalent to typing in the name of the instance manually-no special consideration is given to the selected instance. If there is another instance with the same name, MARVEL will discriminate among them via the scoping rules, regardless if one of them is selected using the mouse.

Similarly, rule names can also be entered by clicking on the rule name. While prompted for input, the user can use the left mouse button to choose one of the displayed rules. If the strategies menu is displayed, or if the required rule is not displayed,

the user can use the right button to select a rule menu by selecting a strategy or clicking 'Go Back' to return to the strageties menu.

We now go to describing the individual functions in the `xface` module.

## 5.1 Initializing the X Interface

The following routines are involved with first time only initialization of the X interface. They are found in the file `xface.c`.

### 5.1.1 CreateStaticMenus()

```
void CreateStaticMenus(controlList)
ControlPtr *controlList;
```

### 5.1.2 SetMainMenuItems()

```
void SetMainMenuItems(items)
PaletteItem items[];
```

### 5.1.3 SetOptionMenuItems()

```
void SetOptionMenuItems(theOptMenu, items, numOfItems)
OPT_MENU_ENTRY  theOptMenu[];
PaletteItem     items[];
int            *numOfItems;
```

### 5.1.4 clear_disp_pixmap()

```
int clear_disp_pixmap()
```

Clear the pixmap that is used for the display.

### 5.1.5 display_entire_pixmap()

```
int display_entire_pixmap()
```

Display the pixmap in the display window.

### 5.1.6 start_marvel_xface()

```
int start_marvel_xface()
```

Start up the marvel x interface. This does lots of goodies, like open the display, create all the basic static windows, set up the fonts as defined in the variables marvel knows about, and so forth.

In theory, if something does not work, the standard interface is reverted to.

## 5.2 Browsing

The browser is contained within the file browse.c.

### 5.2.1 browse_cmd()

```
/*ARGSUSED*/
int browse_cmd(argc, argv)
int argc;
char **argv;
```

This is the main calling routine for the various parts of the browser. This routine is only accessible to the x interface, thus argc and argv are essentially unused.

Unlike other commands, since the browse is a graphics only command, the opts handlers below do all the work, rather then messing with argc and argv.

### 5.2.2 browse_button_zoom()

```
int browse_button_zoom(x,y)
int x,y;
```

This routine will be called by the menu handler whenever the middle button is pressed in the display window while in the browser. It will zoom in on whatever instance is there, unless the the root instance was selected, in which case it will zoom out.

x,y - coordinates where the button was pressed.

### 5.2.3 browse_opts_done()

```
int browse_opts_done()
```

Get browse done options.

### 5.2.4 browse_opts_info()

```
int browse_opts_info()
```

Get browse options.

### 5.2.5 browse_opts_pan()

```
int browse_opts_pan()
```

Panning uses the entire_graphic_info[] table to figure out what the neighbors are. Find the instance that the user clicked on, get its global location, find the appropriate neighbor (+1 for right, -1 for left), get the instance pointer for that neighbor from the table, and call disp_obj() with that instance.

### 5.2.6 browse_opts_zoomin()

```
int browse_opts_zoomin()
```

Get browse zoomin options.

### 5.2.7 browse_opts_zoomout()

```
int browse_opts_zoomout()
```

Get browse zoomout options.

### 5.2.8 BrowseOptsButtons()

```
void BrowseOptsButtons(theEvent)
XEvent *theEvent;
```

First we need to figure out which window the event came from. If it came from the menu window, we need to deal with the pick. If it came from the rule menu, we need to deal with that, too.

### 5.2.9 get_instance_from_coord()

```
INSTANCE_PTR get_instance_from_coord(graphic_table, real_x, real_y,
                                     return_level, return_location)
GRAPHIC_INFO graphic_table[];
int real_x, real_y, *return_location, *return_level;
```

Given x and y coordinates on the display window, find the appropriate instance to which this pick belongs. The algo is as follows:

0. Error check the coordinates, to be sure the pick worked, otherwise we could get a core dump.

```
1.  check if the pick in in the borders, and if so, adjust to be
just past them.

2.  take trunc((real_y + Y_SPACE/2 - HORIZ_BORD) * Y_SPACE +
HORIZ_BORDER as the y coord to look for.

3.  Find the instance from the real_x coordinate using
get_inst_from_coord() return that instance.

graphic_table is the graphic table to use when looking up.  This is
typically graphic_info, which means 'use the current screen
information.'  The other option is 'entire_graphic_info', which
contains the information for the entire object base.

x, y are the x, y coordinates in question.  They are typically
found by a mouse location query.

return_location is the location in the array of the inst's owner
class where to find the instance in question.
```

The following functions are for zooming and panning.

### 5.2.10 pan_opts_left()

```
/*ARGSUSED*/
int pan_opts_left(inst, loc)
INSTANCE_PTR inst;
int loc;


get pan left options.
```

### 5.2.11 pan_opts_right()

```
/*ARGSUSED*/
int pan_opts_right(inst, loc)
INSTANCE_PTR inst;
int loc;


  fillmein
```

### 5.2.12 pan_root_left()

```
/*ARGSUSED*/
int pan_root_left(x,y)
int x,y;
```

Get pan left options.

### 5.2.13 pan_root_right()

```
/*ARGSUSED*/
int pan_root_right(x,y)
int x,y;
```

get pan right options.

The following functions set the validity of the display.

### 5.2.14 valid_display()

```
int valid_display()
```

Check to see if the display is valid.

### 5.2.15 set_display_invalid()

```
void set_display_invalid()
```

Set the val_display flag to FALSE, to specify that the display needs updating.

### 5.2.16 set_display_valid()

```
/*ARGSUSED*/
```

```
void set_display_valid(y_space, num_levels)
int y_space, num_levels;
```

Set the flag that specifies that the display is valid.

## 5.3 System Messages

All system messages pass though routines in the X interface, for consistency. This entanglement is at first not clear, but allows the application level person to write much clearer code. In these routines, the type of interface is determined, and the appropriate low level messages are printed out.

Many of the key message routines are macros, defined in message.h in the include directory. Reference can be found to these in section 3.5. That section should be consulted prior to reading this section, for clarity.

### 5.3.1 message()

```
void message(buf)
char *buf;
```

message() is the main method of printing out strings in marvel. It goes to the next line, and prints out the specified string. Printf should never be used. To get a later message on the same line, use c_message() below.

If you have arguments, use the macro arg_message(). It's use is funny because it is a macro, it is used as follows:

```
arg_message((Mbuf, <normal guts of printf>));
```

Note the important double parens. There is, of course, an c_arg_message() marcro.

message.h must be included.

In message, buf is a NULL terminated character string. If buf is NULL a blank line is printed.

### 5.3.2 init_message()

```
int init_message()
```

Initialize counters for printing in the text window. This should probably be done per command. In the graphics interface, it clears the text window, and in the line interface, it prints a blank line.

### 5.3.3 draw_text_message()

```
/*ARGSUSED*/
static void draw_text_message(buf, start_x)
char *buf;
int start_x;
```

buf is a char pointer to the message.
start_x is the x starting coordinates of the message.

This routine draws a message in the text window. It is only called by the message() routine, that is the normal interface which is used.

### 5.3.4 c_message()

```
void c_message(buf)
char *buf;
```

c_message() continues a previous message, printed with c_message or message(). There is a macro c_arg_message() to continue a message with printf like arguments. See the usage for message() for more details.

### 5.3.5 CreateNewMessageBuffer()

```
Circular_Buffer_Head_Ptr
CreateNewMessageBuffer(maxSlots, maxDisplayable)
     int maxSlots;
     int maxDisplayable;
```

CircularBuffer stuff.

### 5.3.6 DrawTextWindow()

```
void DrawTextWindow()
```

Redraw the TextWindow. Deal with scrolling, and all the nonesense.

### 5.3.7 PageTextWindDown()

```
/*ARGSUSED*/
int
PageTextWindDown(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
```

Page the TextWindow up (one whole screen's worth, that is.

### 5.3.8 PageTextWindUp()

```
/*ARGSUSED*/
int
PageTextWindUp(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
```

Page the TextWindow up (one whole screen's worth, that is.

### 5.3.9 ScrollTextWindDown()

```
/*ARGSUSED*/
int
ScrollTextWindDown(whichControl, whichPart)
     ControlPtr     whichControl;
```

```
     ControlPartPtr whichPart;
```

Scroll the TextWindow down.

### 5.3.10 ScrollTextWindTo()

```
int
ScrollTextWindTo(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
```

Scroll to a particular place. Currently only used for the TextWindow, but it could be used for other scrolling windows.

### 5.3.11 ScrollTextWindUp()

```
/*ARGSUSED*/
int
ScrollTextWindUp(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
```

Scroll the TextWindow up.

### 5.3.12 get_start_coord()

```
void get_start_coord(x, y)
int *x, *y;
```

Get the starting coordinates for a message. Note that we derive them from the location of the last message.

The args are pointers to ints, so both values are returned.

### 5.3.13 start_jump_scroll()

```
start_jump_scroll()
```

```
Turn on jump scrolling.  Jump scrolling should be used whenever a
lot of text is going to be printed out at once.  It should be turned
off with stop_jump_scroll().
```

### 5.3.14 stop_jump_scroll()

```
stop_jump_scroll()
```

```
Turn off jump scrolling.  Jump scrolling should be used whenever a
lot of text is going to be printed out at once.  It should be turned
on with start_jump_scroll().
```

### 5.3.15 draw_message_window()

```
static void draw_message_window(old_first)
int old_first;
```

```
Redraw the entire message window.  Used for jump scrolling and when
a whole page needs to move up.
```

The following are outdated and should not be used. The last four come from page.c.

### 5.3.16 ui_message()

```
void ui_message(buf)
char *buf;
```

```
This one is here for historic reasons only.  Users should use
message().  It will go away as soon as it is not being used.  There is
```

also a ui_arg_message() macro, which should be just a arg_message().

### 5.3.17 text_paging_off()

```
void text_paging_off()
```

This routine turns off text paging. This is an outdated system and should not be used.

text_paging_on() if the complementary one.

### 5.3.18 text_paging_on()

```
void text_paging_on()
```

This routine turns on text paging. This is an outdated system and should not be used.

text_paging_off() if the complementary one.

### 5.3.19 done_with_opts()

```
int done_with_opts()
```

Return TRUE unconditionally, without doint any work.

### 5.3.20 page_output()

```
void page_output(fp)
FILE *fp;
```

This is the head of the paging code for the text window. The basic idea is to write a bunch of output to a text file, then put the file up a page at a time on the screen. To make it quick, an array of file

pointers which point into the file are kept, thus making quick work of searching around in the file when going backwards, or when going forwards after the first time. fp should be an open file descriptore, pointing to the begining of a text file to be output.

### 5.3.21 page_to_next()

```
int page_to_next(num_lines)
int num_lines;
```

find the correct file pointer, and display the page starting from there. Be careful to copy the file pointer before using it, in order to avoid it's becoming corrupted.

num_lines is the number of lines to display.

### 5.3.22 page_to_prev()

```
int page_to_prev(num_lines)
int num_lines;
```

go to the previous page of output. If there is no previous page, then just emit a beep. Note that we need to reset the file pointer in the zero'th element of the file pointer array. This is somewhat mystical, but it gets mangled.

num_lines is the number of lines to display.

## 5.4 Text String Input

Text string input is handled in the following code. It is in `xgetstr.c`

### 5.4.1 draw_cursor()

```
void draw_cursor(erase, x, y)
int erase, x, y;
```

draw a cursor.

Erase is TRUE if the action is removing a cursor, or FALSE to put on a cursor.

x, y are the lower left corners of the cursor.

### 5.4.2 get_str()

```
char *get_str(blanks_ok)
int blanks_ok;
```

Get a character string from the user.

For the normal user interface, just use gets. The parameters are dummies.

Otherwise this routine will use the current location of the text window to do all the output. To find out where to start the string, get_start_coord() is called to get the info from the message package. For the x user interface the process is as follows.

1. lock the keyboard to the passed window.

2. look for key press events.

3. Translate each event into a character. This step is machine dependent based on the mapping in xkeys.h. This file should be ifdefed as other hardware platforms and mappings are added to the system.

4. As characters are being read, display the string, character by character. Handle back spaces, control x (kill), and a few others.

blanks_ok -- TRUE if you want to allow blanks, FALSE to ignore them.

### 5.4.3 handle_key_press()

```
handle_key_press(ch, len, buf, blanks_ok, loc, start_x, start_y)
char *ch, *buf;
```

```
int blanks_ok, *loc, start_x, start_y;
```

This is the main routine that process key input in the X interface.

## 5.5 Displaying the Objectbase

The recursive objectbase display routines are in the file disp_ob.c.

### 5.5.1 calc_nodes_and_hierarchy()

```
static void calc_nodes_and_hierarchy(inst, level, graphic_table)
INSTANCE_PTR inst;
int level;
GRAPHIC_INFO graphic_table[];
```

do all the calculations (recursively), to assure a correct display.
This routine is not available to general users.

To handle multiple classes at the same level, there is a seperate
array which contains pointers to instances at each particular level,
regardless of to which class they belong.

inst -- the inst being worked on now.
level -- the level of recursion we are at. The first level is zero.
graphic_table -- the graphics table that should be operated on.

### 5.5.2 disp_obj()

```
int disp_obj(class, inst)
CLASS_PTR class;
INSTANCE_PTR inst;
```

graphically display the objectbase. Start with a particular
instance of a given class, or start with an entire class. For the
case of displaying an entire class, that class is considered the top
level, and its global_inst_list is traversed to get the whole class.
Note that all other "lower" classes are traversed by their regular
inst_lists. If class is NULL, use inst. One or the other should be NULL.

```
Note that the first level of the tree (the root(s) is level 1, NOT level
0.

6/29/89
disp_obj now writes the display into a pixmap, for fast exposures
when necessary.
```

### 5.5.3 draw_hierarchy()

```
static void draw_hierarchy(class, inst, px, py, graphic_table)
CLASS_PTR class;
INSTANCE_PTR inst;
int px, py;
GRAPHIC_INFO graphic_table[];


actually do the drawing.  If class is not null,
do individual draw hierarchy() calls on each of the instances in that
classes global instance list.

The only reason we need to do this hierarchically, rather then use the
info now in each of the classes, is to get the parent coordinates which
are used to draw connecting lines.  This info could be recovered via
the owner_att->owner_class links, but this might be just as slow.  We
will see.   In general, users should use disp_obj() below.

class -- a class, of which all its instances will be drawn as the top
         level dudes.
inst -- if class is NULL, the root class to put up on the display.
px -- parent's x coordinate.  0 for the top level.
py -- parent's y coordinate.  0 for the top level.
```

### 5.5.4 fill_graphics_table()

```
int fill_graphics_table()


This routine fills up the global full_graphic_info[] table.  This
table contains entries for the entire objectbase, not just what is
displayed on the screen.  It is used for panning left and right, and
zooming in and out.
```

This routine must be called in order to properly recalculate the entire display. Usually, the process is:

1) Calling fill_graphics_table();
2) Calling disp_obj() with the appropriate instance/class (possibly derived from get_graphic_root_inst() ).

This is basically the same as disp_obj(), except that the entire display is hardcoded in, i.e., class = get_db_root(), inst = NULL;

### 5.5.5 get_full_display_status()

```
int get_full_display_status()
```

### 5.5.6 get_graphic_inst_loc()

```
INSTANCE_PTR get_graphic_inst_loc(graphic_table, level, loc)
int level, loc;
GRAPHIC_INFO graphic_table[];
```

This routine returns the instance pointer given in the graphic table. graphic table specifies the graphics table to access. Currently there are only two, entire_graphic_info, which specifies the graphic table for the entire object base, and graphic_info, which specifies the graphic table for the stuff currently displayed. level represents the equivalent of the y coordinate, while loc specifies the x coordinate.

### 5.5.7 get_graphic_root_inst()

```
INSTANCE_PTR get_graphic_root_inst()
```

## 5.6 Finding the Current Object

Is this right, mike? file cur_disp.c.

### 5.6.1 get_current_display_root_class()

```
CLASS_PTR get_current_display_root_class()
```

Get the current root class in the display.

### 5.6.2 get_current_display_root_inst()

```
INSTANCE_PTR get_current_display_root_inst()
```

Get the current root instance in the display.

### 5.6.3 set_current_display_root()

```
void set_current_display_root()
```

set teh current display root class and instance.

## 5.7 Text Scrolling

Text scrolling is accomplished via a circular buffer of text messages ...

### 5.7.1 Add_Item_To_Circular_Buffer()

```
int Add_Item_To_Circular_Buffer(the_buffer, new_item)
     Circular_Buffer_Head_Ptr the_buffer;
     Void_Ptr                 new_item;
```

Puts item supplied into the next slot in the given circular

```
buffer.  Update first & last indices, total size, and display
pointers (if  necessary). The index that the item was inserted at is
returned.


Entry:

  the_buffer - pointer to the circular buffer's header
  new_item   - pointer to the item to be put into the buffer

Exit:
  return value - index into the buffer where the item was put
```

### 5.7.2 Adj_Index()

```
int Adj_Index(the_buffer, item_index)
     Circular_Buffer_Head_Ptr the_buffer;
     int                      item_index;

   return index of item relative to the first item in the buffer.

   Entry:

     the_buffer   - the buffer to look at
     item_index   - the index to adjust

   Exit:

     return value - the adjusted index
```

### 5.7.3 Get_Item_From_Circular_Buffer()

```
Void_Ptr Get_Item_From_Circular_Buffer(the_buffer, which_item)
     Circular_Buffer_Head_Ptr the_buffer;
     int which_item;

   Gets specified item from it's slot in the given circular buffer.
   A generic pointer to the item is returned.

   Entry:
```

```
     the_buffer - pointer to the circular buffer's header
     which_item - index of the item to be retrieved from the buffer

   Exit:

     return value - generic pointer to the item retrieved, or NULL if
                    index supplied exceeds buffers bounds.
```

### 5.7.4 New_Circular_Buffer()

```
Circular_Buffer_Head_Ptr New_Circular_Buffer(num_of_slots,
                                  slot_size, max_displayable)
     int num_of_slots;
     int slot_size;
     int max_displayable;

   Creates a new circular buffer with the supplied number of slots
each of the given size.
A pointer to the newly created buffer header is returned.

Entry:

num_of_slots -    number of item slots to be in new buffer
slot_size -    size of each item to be put into the buffer
max_displayable -  maximum to be displayed at a time

Exit:
return value - pointer to the new circular buffer's header
```

### 5.7.5 Replace_Item_In_Circular_Buffer()

```
void Replace_Item_In_Circular_Buffer(the_buffer, position, new_item)
     Circular_Buffer_Head_Ptr the_buffer;
     int                      position;
     Void_Ptr                 new_item;

    Replace item supplied into the specified slot in the given
circular buffer. Do not change any of the buffer's pointers.


Entry:
```

```
  the_buffer - pointer to the circular buffer's header
  position   - index of item to be replaced
  new_item   - pointer to the item to be put into the buffer

Exit:
  return value - void
```

### 5.7.6 Reset_Circ_Buff_Display_Ptrs()

```
void Reset_Circ_Buff_Display_Ptrs(the_buffer)
     Circular_Buffer_Head_Ptr the_buffer;


   Resets the buffer's display pointers to the first display page
   of the buffer.

   Entry:

     the-buffer   - the buffer to be reset

   Exit:
     return value - NONE
```

### 5.7.7 Scroll_Circular_Buffer()

```
void Scroll_Circular_Buffer(the_buffer, how_much)
     Circular_Buffer_Head_Ptr the_buffer;
     int how_much;

     Scroll the circular buffer's display pointers.

   Entry:

     the_buffer     - pointer to the circular buffer's header
     how_many_lines - how many lines up or down to scroll the circular
                      buffer's display pointers. A negative value means
      that the text is scrolled back, otherwise forwards

   Exit:
```

```
     return value   - NONE
```

### 5.7.8 Scroll_To()

```
void Scroll_To(the_buffer, position)
     Circular_Buffer_Head_Ptr the_buffer;
     int position;


   Scrolls circular buffer's display pointers to start at the specified
   item.

   Entry:

     the-buffer   - the buffer to be reset

   Exit:
     return value - NONE
```

The code for this is found in `circ_buff.c`

## 5.8 Menu Handling

These functions, found in `menu.c`, manage the various menus in MARVEL . These are higher level routines, others are found in different parts of the X interface.

### 5.8.1 draw_menu()

```
void draw_menu()

   put up the main static menu for built in marvel commands.
```

### 5.8.2 draw_opt_menu()

```
void draw_opt_menu(theOptMenu)
     ControlPtr theOptMenu;
```

opt_menu_def is a pointer to an array of options for some appropriate circumstance. All the various opt menus are currently defined in interpreter/cmd_defs.c.

num_entrys is the number of entries in this menu. It would be more ultimate to have a special end keyword at the end of each menu, so they could be sort of dynamic in length. This should change at some point.

### 5.8.3 draw_rule_menu()

```
void draw_rule_menu()
```

Draw the rule menu.

### 5.8.4 OptionEvents()

```
void
OptionEvents(display, theEvent, otherEventMask, buttonEventHandler)
     Display          *display;
     XEvent           *theEvent;
     unsigned long     otherEventMask;
     void             (*buttonEventHandler)();
```

Deal with events while in an option.

### 5.8.5 clear_opt_menu()

```
void clear_opt_menu(theOptMenu)
     ControlPtr theOptMenu;
```

clear the opt_menu window. Remember that since this menu overlays the menu window, that one must be redisplayed.

### 5.8.6 handle_menu_pick()

```
int handle_menu_pick(theEvent)
     XEvent *theEvent;
```

This routine takes care of menu picks for the from the main command menu. It just creates a simple command line, and calls the standard entry point expand_and_execute().

Note that there is a potential problem here, because the menu represents a specific command, but really allt that the expander gets is a string that it then tries to expand. We should probably have a direct connection to the proper function to be called for this interface, rather then go through this. This might alleviate some problems with rules being named the same as menu items.

### 5.8.7 handle_menu_pick_no_execute()

```
int handle_menu_pick_no_execute()
```

this routine is called when a command
needs to use another command's name as part of the input. After
accepting a pick, it will just return the slot of the thing picked,
without executing it.

### 5.8.8 handle_opt_pick()

```
int handle_opt_pick(whichMenu, optionEvents, optionEventsMask, buttonEvents)
     ControlPtr      whichMenu;
     void           (*optionEvents)();
     unsigned long   optionEventsMask;
     void           (*buttonEvents)();
```

Handle the picking of an option from the specified option menu (in whichMenu).

### 5.8.9 handle_rule_menu_pick()

```
int handle_rule_menu_pick(theEvent, execute)
XEvent *theEvent;
int     execute;
```

theEvent is the initial event that occured in the rule menu.
If execute is TRUE, then the rule chosen is executed, otherwise not.

### 5.8.10 set_rule_menu_invalid()

```
void set_rule_menu_invalid()
```

Set a flag that specifies that the rule menu is invalid.

### 5.8.11 set_rule_menu_valid()

```
/*ARGSUSED*/
void set_rule_menu_valid()
```

Set a flag that specifies that the rule menu is valid.

### 5.8.12 set_up_rule_table()

```
int set_up_rule_table()
```

This routine fills up the rule menu table. This table contains
information regarding what strategies or rules are currently
active. This should be called whenver the rule menu is changed in
any way..

### 5.8.13 string_in_strlist()

```
int
```

```
string_in_strlist(list, string)
STRLIST_PTR list;
char *string;
```

This routine checks a strlist to see if a given string is in it.

### 5.8.14 valid_rule_menu()

```
int valid_rule_menu()
```

Check to see if the rule menu is valid.

## 5.9 Graphic Object Control

Mike T, what is the scoop.

### 5.9.1 ActivateControl()

```
void
ActivateControl(theControl)
     ControlPtr theControl;
```

Input: theControl -- the control to activate

Output: VOID

Description:

Activate a given control. If the control was not already active, call control's UPDATE function and the its DRAW function, if necessary.

### 5.9.2 AddControlPart()

```
void
AddControlPart(theControl, thePart)
     ControlPtr     theControl;
     ControlPartPtr thePart;
```

Input: theControl -- the control to add the part to.

```
       thePart     -- the part to add to the control.

Output: VOID

Description:

Add a control part to the end of a given control's parts list.
```

### 5.9.3 AppendControlList()

```
void
AppendControlList(controlList, theControl)
     ControlPtr *controlList;
     ControlPtr  theControl;

Input: controlList  -- the list of controls to add the given
                       control to
       theControl   -- the control to add to the control list

Output: VOID

Description:

Add the specified control to the given control list. Used internally
by NewControl.
```

### 5.9.4 ButtonInMask()

```
Boolean
ButtonInMask(whichButton, keys_buttons)
     unsigned int whichButton;
     unsigned int keys_buttons;
```

### 5.9.5 ControlHit()

```
ControlPtr
ControlHit(controlList, theEvent)
     ControlPtr  controlList;
     XEvent     *theEvent;
```

```
Input: controlList -- list of controls to consider
       theEvent    -- pointer to the XEvent structure

Output:

RETURN -- the control that event occurred in, if any, else NIL

Description:

Test if theEvent occurred in any active control. If so, the control
referenced is returned. Otherwise NIL is returned
```

### 5.9.6 ConvertSlidePosToValue()

```
int ConvertSlidePosToValue(whichControl, whichPart)
ControlPtr     whichControl;
ControlPartPtr whichPart;


Input: whichControl  -- the control containing the sliding part
       whichPart     -- the sliding part

Output: RETURN       -- the value corresponding to the position of
                        the part
Description:

Convert the position of the sliding part to be a discrete value
between the control's minimum and maximum values.
```

### 5.9.7 CreateControlGC()

```
void
CreateControlGC(display, d)
     Display  *display;
     Drawable  d;

Input: display  -- pointer to X Display structure
       d        -- drawable (usually: ROOT window/pixmap) to use for
                   GC creation
Output: CONTROL_MANAGER_GC  -- GLOBAL GC for control manager use only

Description:
```

```
Create global control manager graphics context, using built-in
(hard-coded) default values. Used internally by InitControls.
```

### 5.9.8 DeactivateControl()

```
void
DeactivateControl(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to deactivate

Output: VOID

Description:

Deactivate a given control. If the control was active, call
control's UPDATE function and the its DRAW function, if necessary.
```

### 5.9.9 DisposeControl()

```
void
DisposeControl(controlList, theControl)
     ControlPtr *controlList;
     ControlPtr  theControl;

Input: controlList  -- the list of controls
       theControl   -- control to remove

Output: VOID

Description:

Remove a control from the control list and reclaim its space in memory.
```

### 5.9.10 DoControl()

```
int
DoControl(whichControl, theEvent)

     ControlPtr  whichControl;
     XEvent     *theEvent;
```

```
Input: whichControl  -- control that mouse button was pressed in
       theEvent      -- the initial mouse button down event

Output: RETURN: partNumber of part hit,or NO_PART

Description:
```

This function is called after a mouse button pressed event is received from within the bounds of a control window. At this point, the control window has performed a grab of the mouse, and will have the mouse until the depressed button is released. We all sympathize with this poor depressed button, don't we? In any case, the sequence of steps taken by this function are as follows:

1) Highlight the part of the control the pointer was in at the time of the button-pressed event.
2) perform the buttDownFunc for the selected control part, if any.
3) if control part is a sliding part, continue processing with the function TrackSlideControl. Otherwise part is non-moving type of part (static), and is processed by TrackStaticControl. The value returned by TrackStaticControl or TrackSlideControl is the value returned by this function.

An example of how to use controls in the event handling module of a program would be (assuming that the controls have all been created):

```
...

XNextEvent(display, &xevent);
switch (xevent.type) {
case ButtonPress:

  if (whichControl = ControlHit (controls, &xevent))
    partNumber = DoControl (whichControl, &xevent);
  else
    HandleOtherButtonPresses (&xevent, ...);
  break;

case ...:
case ...:
}

...
```

### 5.9.11 DrawButton()

```
/*ARGSUSED*/
void
DrawButton(theButton, drawAllParts)
     ControlPtr theButton;
     Boolean    drawAllParts;
```

### 5.9.12 DrawControl()

```
void
DrawControl(theControl, drawAllParts)
     ControlPtr theControl;
     Boolean    drawAllParts;

Input: theControl    -- the control to redraw
       drawAllParts  -- TRUE if should redraw all of the control's
                        parts, regardless of the value of the part's
                        mustRedrawPart flag.
Output: VOID

Description:

Calls the given control's drawControl function, if any.  Otherwise, calls
DrawPart for each part in the given control's part list.  If the control
is using a pixmap, the pixmap is displayed.
```

### 5.9.13 DrawControlPartOutline()

```
void
DrawControlPartOutline(whichControl, whichPart, displayFunction)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
     int            displayFunction;

Input: whichControl     -- the control containing the part to outline
       whichPart        -- the part to draw an outline of
       displayFunction  -- the display function to specify in the GC
                           when drawing the part's outline

Output: VOID

Description:
```

Draw an outline of the given part's boundary rectangle, using the given
X display function. Generally only called by TrackSlideControl.

### 5.9.14 DrawPagingRegion()

```
void
DrawPagingRegion(theControl, thePagingPart, drawAllParts)
     ControlPtr     theControl;
     ControlPartPtr thePagingPart;
     Boolean        drawAllParts;
```

draw scroll bar paging region.

### 5.9.15 DrawPaletteItem()

```
void
DrawPaletteItem(thePalette, theItem, drawAllParts)
     ControlPtr     thePalette;
     ControlPartPtr theItem;
     Boolean        drawAllParts;
```

### 5.9.16 DrawPart()

```
void
DrawPart(theControl, thePart, drawAllParts)
     ControlPtr     theControl;
     ControlPartPtr thePart;
     Boolean        drawAllParts;

Input: theControl    -- the control to redraw
       thePart       -- the part to draw
       drawAllParts  -- TRUE if should redraw all of the control's
                        parts, regardless of the value of the part's
                        mustRedrawPart flag.
Output: VOID

Description:
```

```
Execute the given part's drawPart function, if any.  Otherwise, default
to just drawing rectangle around perimeter of part, resetting its
mustRedrawPart flag to FALSE.
```

### 5.9.17 DrawScrollArrow()

```
void
DrawScrollArrow(theControl, theScrollArrow, drawAllParts)
     ControlPtr     theControl;
     ControlPartPtr theScrollArrow;
     Boolean        drawAllParts;


Draw scroll bar's scroll arrow.
```

### 5.9.18 DrawScrollableMenuItem()

```
void
DrawScrollableMenuItem(theMenu, theItem, drawAllParts)
     ControlPtr     theMenu;
     ControlPartPtr theItem;
     Boolean        drawAllParts;
```

### 5.9.19 DrawTextCentered()

```
void
DrawTextCentered(theDisplay, theDrawable, theGC, theRect, theText)
     Display    *theDisplay;
     Drawable    theDrawable;
     GC          theGC;
     XRectangle *theRect;
     char       *theText;
```

### 5.9.20 DrawThumb()

```
void DrawThumb(theControl, theThumb, drawAllParts)
```

```
    ControlPtr theControl;
    ControlPartPtr theThumb;
    Boolean drawAllParts;


Draw scroll bar's thumb.
```

### 5.9.21 FigureThumbXPos()

```
int
FigureThumbXPos(theScrollBar, theThumbPart, slideLeftX, slideRightX)
    ControlPtr     theScrollBar;
    ControlPartPtr theThumbPart;
    int            slideLeftX;
    int            slideRightX;
```

### 5.9.22 FigureThumbYPos()

```
int
FigureThumbYPos(theScrollBar, theThumbPart, slideTopY, slideBottomY)
    ControlPtr     theScrollBar;
    ControlPartPtr theThumbPart;
    int            slideTopY;
    int            slideBottomY;
```

### 5.9.23 FindPart()

```
ControlPartPtr
FindPart(whichControl, x, y)
    ControlPtr  whichControl;
    int         x, y;

Input: whichControl  -- the control that contains the parts to check
       x             -- the x coordinate of the mouse click
       y             -- the y coordinate of the mouse click

Output: RETURN        -- the part that the mouse button was clicked in
```

```
Description:

Given an X and Y coordinate defining a point, search the specified
control's parts list for the part containing the point.
```

### 5.9.24 ForceControlInput()

```
int
ForceControlInput(theControl, controls, otherEventFunc, otherEventMask,
            otherButtonEvents, allowOtherControls)
     ControlPtr      theControl;
     ControlPtr      controls;
     void           (*otherEventFunc) ();
     unsigned long   otherEventMask;
     void           (*otherButtonEvents) ();
     Boolean         allowOtherControls;

Input: theControl          --  the control requiring user input
       controls            --  the list of all controls
       otherEventFunc      --  a function to perform if an event occurs
                               outside
                               of a control, NIL if no others allowed
       otherEventMask      --  the event mask to be passed to
                               otherEventFunc
       otherButtonEvents   --  the function to process mouse button
                               events, passed to otherEventFunc
       allowOtherControls  --  TRUE if controls other than theControl
                               are allowable
Output: RETURN  -- the part number of the part of the control hit

Description:

Force the user to hit a specific control. Do not return until the user
selects a part of that control.  Process other events, if otherEventFunc
is specified, and process other control events if allowOtherControls is
TRUE.  Return part number of theControl which was selected.
```

### 5.9.25 FreeControl()

```
void
FreeControl(theControl)
     ControlPtr theControl;
```

```
Input: theControl  -- pointer to the control structure to free

Output: VOID

Description:

Free the memory used by a specified control structure. USED INTERNALLY
by DisposeControl.
```

### 5.9.26 FreeControlPart()

```
void
FreeControlPart(thePart, freePart)
     ControlPartPtr thePart;
     void (*freePart) ();

Input: thePart  -- the part to free
       freePart -- the function to use to free part, if any

Output: VOID

Description:

Free a specific control part. If the freePart parameter is specified
as a non-NIL value, then use it as the function to free the given part.
If freePart is NIL, then use the default FREE function.
```

### 5.9.27 FreeControlParts()

```
void
FreeControlParts(partsList, freePart)
     ControlPartPtr partsList;
     void (*freePart) ();

Input: partsList  -- list of control parts to free
       freePart   -- function to use to free one control part

Output: VOID

Description:
```

```
Deallocate memory used by control parts in given parts list.
```

### 5.9.28 GetControlMaxVal()

```
int
GetControlMaxVal(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to retrieve val from

Output: RETURN -- the control's maximum allowable value

Description:

Get the maximum allowable value for a control.
```

### 5.9.29 GetControlMinVal()

```
int
GetControlMinVal(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to retrieve val from

Output: RETURN -- the control's minimum allowable value

Description:

Get the minimum allowable value for a control.
```

### 5.9.30 GetControlVal()

```
int
GetControlVal(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to retrieve val from

Output: RETURN -- the control's current value

Description:

Get the current value of a control.
```

### 5.9.31 GetEventCoords()

```
void
GetEventCoords(theEvent, x, y)
     XEvent  *theEvent;
     int     *x;
     int     *y;
```

### 5.9.32 GetNextControlEvent()

```
Boolean
GetNextControlEvent(whichControl, theEvent)
     ControlPtr   whichControl;
     XEvent     **theEvent;


Input: whichControl  -- the control to get the next event for
       theEvent      -- pointer to pointer to event structure storage

Output: theEvent      -- the event retrieved, if any
        RETURN        -- TRUE if any new events retrieved

Description:

Get the next event from the event queue, if any.  If more than one
ButtonMotionEvent waiting, get them all to catch display up to
user's movements
```

### 5.9.33 GetWindowRect()

```
void
GetWindowRect(display, window, returnRect)
     Display    *display;
     Window      window;
     XRectangle *returnRect;
```

### 5.9.34 HideControl()

```
void
HideControl(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to hide.

Output: VOID

Description:

Unmap and deactivate a control. If control already hidden, has no effect.
```

### 5.9.35 HighlightPart()

```
void
HighlightPart(whichPart)
     ControlPartPtr whichPart;

Input: whichPart  -- the control part to highlight

Output: VOID

Description:

Highlight a control part, setting its redisplay flag to TRUE.
```

### 5.9.36 InitControls()

```
void
InitControls(display, d, defaultGC, fontName)
     Display  *display;
     Drawable *d;
     GC       *defaultGC;
     char     *fontName;

Input: display   -- pointer to X Display structure
       d         -- drawable (usually: ROOT window/pixmap) to use for
                    GC creation
       defaultGC -- either GC to duplicate for use or NIL to use
                    built-in default
       fontName  -- name of font to use (which MUST exist on system or
```

```
                    will exit)

Output: CONTROL_MANAGER_GC          -- GLOBAL GC for control manager
                                       use only
        CONTROL_MANAGER_FONT_STRUCT -- GLOBAL font struct for control
                                       mgr use only
Description:

Initialize control manager by creating global structures.
```

### 5.9.37 InsetRect()

```
void
InsetRect(theRect, howMuch)
     XRectangle *theRect;
     int         howMuch;


Shrink the given rectangle (destructively modify) by howMuch (in
pixels) in all directions, without moving the rectangle's center.
```

### 5.9.38 IsControlActive()

```
Boolean
IsControlActive(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to test

Output: TRUE if the given control is active, otherwise FALSE

Description:

Test if a given control is active.
```

### 5.9.39 IsHighlighted()

```
Boolean
IsHighlighted(whichPart)
     ControlPartPtr whichPart;

Input: whichPart  -- the control part to check
```

```
Output: RETURN  -- TRUE if part is highlighted, otherwise FALSE.

Description:

Test whether a control part is currently highlighted.
```

### 5.9.40 IsOKTimeToRepeat()

```
/*ARGSUSED*/
Boolean
IsOKTimeToRepeat(thePart)
     ControlPartPtr thePart;


supposed to return TRUE if the part's repeatDelay time has past, but
currently always returns TRUE
```

### 5.9.41 IsPartHit()

```
Boolean
IsPartHit(presPart, x, y)
     ControlPartPtr  presPart;
     int             x, y;

Input: presPart  -- the part to check
       x         -- the x coordinate of the mouse button press
       y         -- the y coordinate of the mouse button press

Output: RETURN   -- TRUE if point where mouse down event occurred
                    is within the bounds of the specified control part,
                    else FALSE.
Description:

Return TRUE if the given event occurred in the given control part,
otherwise FALSE.
```

### 5.9.42 MakeScrollDownData()

```
ScrollArrowDataPtr
MakeScrollDownData(partRect)
     XRectangle *partRect;
```

### 5.9.43 MakeScrollLeftData()

```
ScrollArrowDataPtr
MakeScrollLeftData(partRect)
     XRectangle  *partRect;
```

### 5.9.44 MakeScrollRightData()

```
ScrollArrowDataPtr
MakeScrollRightData(partRect)
     XRectangle *partRect;
```

### 5.9.45 MakeScrollUpData()

```
ScrollArrowDataPtr
MakeScrollUpData(partRect)
     XRectangle *partRect;
```

### 5.9.46 NewButton()

```
ControlPtr
NewButton(display, parent, controlList, x, y, width, height, name,
          active, data, buttonAction)

     Display      *display;        connection to the X server              */
     Window        parent;         parent window ID                        */
     ControlPtr   *controlList;    list of all controls parent window has  */
     int           x;              top left x&y coords of this button      */
     int           y;                relative to it's parent window        */
     unsigned int  width;          width (in pixels) of button             */
```

```
    unsigned int  height;         height (in pixels) of button          */
    char         *name;           the name of this button               */
    Boolean       active;         TRUE if new button is to be active    */
    physadr       data;           whatever                              */
    int          (*buttonAction)(); function to exec if button selected  */
```

## 5.9.47 NewControl()

```
ControlPtr
NewControl(display, parent, controlList, x, y, width, height, border_width,
           cursor, name, whatType, minValue, maxValue, currentValue, active,
           usePix,data,updateFunc, drawFunc, freeControlData, freeControlPart)
    Display       *display;
    Window         parent;
    ControlPtr    *controlList;
    int            x;
    int            y;
    unsigned int   width;
    unsigned int   height;
    unsigned int   border_width;
    Cursor         cursor;
    char          *name;
    ControlType    whatType;
    int            minValue;
    int            maxValue;
    int            currentValue;
    Boolean        active;
    Boolean        usePix;
    physadr        data;
    Boolean      (*updateFunc) ();
    void         (*drawFunc) ();
    void         (*freeControlData) ();
    void         (*freeControlPart) ();

Input: Display       *display;      connection to the X server
       Window         parent;       parent window ID
       ControlPtr    *controlList;  list of controls parent wind has
       int            x;            top left x&y coords of control
       int            y;              relative to parent window
       unsigned int   width;        control width (not incl border)
       unsigned int   height;       control height (not incl border)
       unsigned int   border_width; border width of control (in pix)
       Cursor         cursor;       the cursor for this control
       char          *name;         the name of this control
       ControlType    whatType;     control type specifier
```

```
        int             minValue;     minimum value of this control
        int             maxValue;     maximum value of this control
        int             currentValue; current value of this control
        Boolean         active;       TRUE if this control is active
        Boolean         usePix;       TRUE if should use pixmap for display
        physadr         data;         whatever

        Boolean       (*updateFunc) (ControlPtr whichControl);
                          the rgn update func

        void          (*drawFunc) (ControlPtr whichControl,
                                   Boolean drawAllParts);
                        Function to draw control. a NULL causes the default
                        draw routine to exec.  The default routine just
                        execs each part in parts-list's own draw function.
                        If drawAllparts is true, all parts are redrawn,
                        regardless of whether or not the part needs it.

        void          (*freeControlData) (void *data);
                        Function to free the data assoc with the control.
                        If NULL, then the standard C library function FREE
                        is used.

      void          (*freeControlPart) (ControlPartPtr whichPart);
                        Function to free an individual control part. if
                        NULL, then standard C library function FREE
                        is used.
```

Output: RETURN -- the new control created, else NIL if error.

Description:

Create new control, but no its parts.

## 5.9.48 NewDownScrollingMenuArrow()

```
ControlPartPtr
NewDownScrollingMenuArrow(x, y, width)
     int          x, y;
     unsigned int width;
```

### 5.9.49 NewMenuPalette()

```
ControlPtr
NewMenuPalette(display, parent, controlList, x, y, width, height,
               border_width, cursor, name, whatType, active, data,
               numOfItems, itemArray)
     Display       *display;
     Window         parent;
     ControlPtr    *controlList;
     int            x;
     int            y;
     unsigned int   width;
     unsigned int   height;
     unsigned int   border_width;
     Cursor         cursor;
     char          *name;
     ControlType    whatType;
     Boolean        active;
     physadr        data;
     int            numOfItems;
     PaletteItem   itemArray[];
```

### 5.9.50 NewPaletteItem()

```
ControlPartPtr
NewPaletteItem(itemSpec, itemNumber, x, y, width, height)
     PaletteItem  itemSpec;
     int          itemNumber;
     int          x;
     int          y;
     unsigned int width;
     unsigned int height;
```

### 5.9.51 NewScrollBar()

```
ControlPtr
NewScrollBar(display, parent, controlList, x, y, width, height,
             name, minVal, maxVal, val, active, data,
             lineUp, lineDown, pageUp, pageDown, gotoLine)
```

```
    Display      *display;        connection to the X server   */
    Window        parent;         parent window ID             */
    ControlPtr   *controlList;    list of all controls parent
                                                 window has    */
    int           x;              top left x&y coords of this
                                                 scroll bar    */
    int           y;                relative to it's parent
                                               window          */
    unsigned int  width;          width (in pixels) of s bar   */
    unsigned int  height;         height (in pixels) of s bar  */
    char         *name;           the name of this scroll bar  */
    int           minVal;         minimum acceptable value for
                                                 scroll bar    */
    int           maxVal;         maximum acceptable value for
                                                 scroll bar    */
    int           val;            initial value for scroll bar */
    Boolean       active;         TRUE if new scroll bar is to
                                                 be active     */
    physadr       data;           whatever                     */
    int          (*lineUp)();     function to move up a line   */
    int          (*lineDown)();   function to move down a line */
    int          (*pageUp)();     function to move up a page   */
    int          (*pageDown)();   function to move down a page */
    int          (*gotoLine)();   func to move to a particular
                                                 line (thumb)  */
```

Create & return pointer to newly created (incl mem allocation)
scroll bar.

### 5.9.52 NewScrollableMenu()

```
ControlPtr
NewScrollableMenu(display, parent, controlList, x, y, width, height,
            border_width, cursor, name, whatType, active,
            numOfItems, itemArray)
    Display       *display;
    Window         parent;
    ControlPtr    *controlList;
    int            x;
    int            y;
    unsigned int   width;
    unsigned int   height;
    unsigned int   border_width;
```

```
    Cursor          cursor;
    char           *name;
    ControlType     whatType;
    Boolean         active;
    int             numOfItems;
    PaletteItem     itemArray[];
```

### 5.9.53 NewScrollableMenuItem()

```
ControlPartPtr
NewScrollableMenuItem(itemSpec, itemNumber, x, y, width, height)
    PaletteItem  itemSpec;
    int          itemNumber;
    int          x;
    int          y;
    unsigned int width;
    unsigned int height;
```

### 5.9.54 NewUpScrollingMenuArrow()

```
ControlPartPtr
NewUpScrollingMenuArrow(x, y, width)
    int          x, y;
    unsigned int width;
```

### 5.9.55 PointInRect()

```
Boolean
PointInRect(theRect, x, y)
    XRectangle *theRect;
    int         x, y;
```

Return TRUE if a given point is within the bounds of a given rectangle.

### 5.9.56 PushControlPart()

```
void
PushControlPart(theControl, thePart)
     ControlPtr     theControl;
     ControlPartPtr thePart;

Input: theControl  -- the control to add the part to.
       thePart     -- the part to add to the control.

Output: VOID

Decription:

Push a control part onto beginning of given control's parts list.
```

### 5.9.57 RemoveFromControlList()

```
void
RemoveFromControlList(controlList, theControl)
     ControlPtr *controlList;
     ControlPtr  theControl;

Input: controlList  -- the list of controls to remove control from
       theControl   -- the control to remove

Output: VOID

Description:

Remove a specified control from the given list of controls. Used
internally by DisposeControl.
```

### 5.9.58 ScrollDownMenu()

```
/*ARGSUSED*/
int
ScrollDownMenu(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
```

### 5.9.59 ScrollUpMenu()

```
/*ARGSUSED*/
int
ScrollUpMenu(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;
```

### 5.9.60 SetControlMaxVal()

```
void
SetControlMaxVal(theControl, newVal)
     ControlPtr theControl;
     int        newVal;

Input: theControl  -- the control to modify
       newVal      -- the new value to assign

Output: VOID

Description:

Set the maximum allowable value for a control.
```

### 5.9.61 SetControlMinVal()

```
void
SetControlMinVal(theControl, newVal)
     ControlPtr theControl;
     int        newVal;

Input: theControl  -- the control to modify
       newval      -- the new value to assign

Output: VOID

Description:

Set the minimum allowable value for a control.
```

### 5.9.62 SetControlVal()

```
void
SetControlVal(theControl, newVal)
     ControlPtr theControl;
     int        newVal;

Input: theControl  -- the control to modify
       newVal      -- the new value to assign

Output: VOID

Description:

Set the current value of a control.
```

### 5.9.63 SetSlideControlFinalPos()

```
void
SetSlideControlFinalPos(whichControl, whichPart)
     ControlPtr     whichControl;
     ControlPartPtr whichPart;

Input: whichControl  -- the control containing the sliding control part
       whichPart     -- the sliding control part

Output: RETURN  -- VOID


Description:

Reset the control's value and then adjust the given sliding part to
be at the position denoting that value.  The control value
is determined by the initial position of the sliding part.
```

### 5.9.64 SetSlideControlPartPos()

```
void
SetSlideControlPartPos(whichPart, x, y, pointerDisplacement)
     ControlPartPtr whichPart;
     int            x;
     int            y;
     int            pointerDisplacement;
```

```
Input: whichPart  -- the control's sliding part which is being moved
       x          -- the current x coordinate of the mouse
       y          -- the current y coordinate of the mouse
       pointerDisplacement
                  -- the distance of the mouse cursor from the top or
                     left edge of the the sliding part, depending on
                     whether the part is a horizontal or vertical
                     slider

Output: RETURN    -- VOID
```

Description:

Move the specified part according to the given x and y coordinates, adjusted by the specified displacement, although the movement is constrained remain within the part's boundaries.

### 5.9.65 ShowControl()

```
void
ShowControl(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to show (unhide)

Output: VOID
```

Description:

Raise and activate a hidden control. If control already shown, has no effect.

### 5.9.66 TrackSlideControl()

```
int
TrackSlideControl(whichControl, whichPart, theEvent)
     ControlPtr      whichControl;
     ControlPartPtr  whichPart;
     XEvent         *theEvent;

Input: whichControl  -- the control containing the part being
```

```
                        referenced
        whichPart       -- the part being referenced
        theEvent        -- the initial event occurring in the control
                           part
Output: RETURN          -- the part the user selected.
```

Description:

While the mouse button is held down, get events, tracking mouse position. If the mouse has moved, draw the part's outline at the new position. The part's whileDownFunc is executed, and if its repeatFlag is true, is continuously executed as the mouse button is held down. When the mouse button is released, the part's buttUpFunc (if any) is executed, and then its final position is calculated. The part's part number is returned.

## 5.9.67 TrackStaticControl()

```
int
TrackStaticControl(whichControl, whichPart, theEvent)
     ControlPtr      whichControl;
     ControlPartPtr  whichPart;
     XEvent         *theEvent;

Input: whichControl  -- the control containing the part being referenced
       whichPart     -- the part being referenced
       theEvent      -- the initial event occurring in the control part

Output: RETURN       -- the part the user selected, or NO_PART.
                        See below.
```

Description:

The algorithm used to process a user's control part selection of a static control part involves two steps:

1) perform the control part's whileDownFunc, if any. Repeat this step if the part's repeatFlag is set, and for as long as the mouse-button remains down and the pointer remains within the bounds of the part, with a time delay of repeatDelay between successive invocations. If the pointer leaves the part while the button is down, unhighlight the part and stop performing the whileDownFunc until the pointer returns to the part.
2) When the mouse button is released, if it is within the bounds of the originally selected part, unhighlight the part and execute it's buttUpFunc, if any, and return the selected part's partNumber. If it is released outside the bounds of the originally selected part,

```
   the this function just returns, with a partNumber of NO_PART (== 0)
```

### 5.9.68 TurnOffPaletteFunctions()

```
void
TurnOffPaletteFunctions(thePalette)
     ControlPtr thePalette;
```

### 5.9.69 TurnOnPaletteFunctions()

```
void
TurnOnPaletteFunctions(thePalette)
     ControlPtr thePalette;
```

### 5.9.70 UnhighlightPart()

```
void
UnhighlightPart(whichPart)
     ControlPartPtr whichPart;

Input: whichPart  -- the control part to unhighlight

Output: VOID

Description:

Unhighlight a control part, setting its redisplay flag to TRUE.
```

### 5.9.71 UpdateControl()

```
Boolean UpdateControl(theControl)
     ControlPtr theControl;

Input: theControl  -- the control to update

Output: RETURN     -- TRUE if should redraw (updating caused a change
```

```
                    in how some part(s) of the control should be
                    displayed).
Description:

Execute the control's update function, if any, returning the value it
returns.  If no update function is specified for this control, then assume
that the control must be redrawn, and return TRUE.
```

### 5.9.72 UpdateHorizontalScrollBar()

```
int
UpdateHorizontalScrollBar(theScrollBar, theUpArrowPart, theDownArrowPart,
                          thePageUpPart, thePageDownPart, theThumbPart)
     ControlPtr     theScrollBar;
     ControlPartPtr theUpArrowPart;
     ControlPartPtr theDownArrowPart;
     ControlPartPtr thePageUpPart;
     ControlPartPtr thePageDownPart;
     ControlPartPtr theThumbPart;
```

### 5.9.73 UpdateScrollBar()

```
Boolean
UpdateScrollBar(theScrollBar)
     ControlPtr theScrollBar;
```

### 5.9.74 UpdateVerticalScrollBar()

```
int
UpdateVerticalScrollBar(theScrollBar, theUpArrowPart, theDownArrowPart,
                        thePageUpPart, thePageDownPart, theThumbPart)
     ControlPtr     theScrollBar;
     ControlPartPtr theUpArrowPart;
     ControlPartPtr theDownArrowPart;
     ControlPartPtr thePageUpPart;
     ControlPartPtr thePageDownPart;
     ControlPartPtr theThumbPart;
```

## 5.10 Painting the Display

Code to paint and repaint the MARVEL display is found in `paint.c`.

### 5.10.1 mouse_pick_from_disp()

```
int mouse_pick_from_disp(x, y)
int *x, *y;

   wait for a Button press window event from the disp window, and then
query the pointer for the coordinates of the pick.

   Care must be taken here when using dbx, because the movement to the
debugging window while stepping through a routine will cause the
coordinates of the query to be different from those of the event, and
you will not get proper results.

   x is a pointer to an integer, in which the x coordinate will be
returned.
   y is a pointer to an integer, in which the y coordinate will be returned.
```

### 5.10.2 mouse_pick_with_done()

```
mouse_pick_with_done(x, y)
int *x, *y;

   mouse_pick_with_done() returns TRUE if the mouse button was pressed
   in the display window, along with the x and y coordinates.  If the
   done menu option was chosen, it returns false.

   This should be used by most of the organ commands, instead of the
   done boxes.
```

### 5.10.3 paint_disp()

```
void paint_disp()
```

The display screen is where the main marvel objectbase is displayed in a tree format.

### 5.10.4 paint_screen()

```
void paint_screen()
```

Paint the screen. Is this used anymore??

### 5.10.5 paint_status()

```
void paint_status(cmd)
char *cmd;
```

The status window is the titlebar at the top of the display window. It is used to identify marvel and the current instance.

### 5.10.6 paint_text()

```
void paint_text()
```

This routine is a noop. The DrawTextWind routines do this complex task now.

## 5.11 Fonts

Font handling code in found in the file `xfonts.c`.

### 5.11.1 set_bold_font()

```
int set_bold_font(font)
```

```
char *font;

   The bold font is not allowed to be a variable sized font, because
   of the graphical input routines.  The best way I see of checking to
   see if a font is variable width font is to check the maximum width
   with the minimum width.  Is there a better way?  Anyway, a -1 is
   returned by set_bold_font if the font is variable sized.
```

### 5.11.2 set_normal_font()

```
int set_normal_font(font)
char *font;

Set the normal font.
```

### 5.11.3 set_small_font()

```
int set_small_font(font)
char *font;

Set the small font.
```

## 5.12 Events

Event handling code in found in the file xevents.c.

### 5.12.1 do_main_loop()

```
int do_main_loop()

   get the thing to display

   wait for window to be drawn on screen before attempting to draw
   on it.  This will become redundant when we have proper exposure
   event handling
```

### 5.12.2 handle_expose_event()

```
handle_expose_event()

    since the menu and rule_menu are completely redrawn every
    single time, it is sufficient to process only one exposure
    event.  The application receives many exposure events for
    every rectangle that was exposed, to ensure that strategies
    like the one for the display window still work.
```

## 5.13 Drawing Links

These functions are used to draw links as arcs in the X-interface.

### 5.13.1 FigureLinkArc()

```
static void
FigureLinkArc(arcPtr, startX, startY, endX, endY)
    XArc *arcPtr;
    int   startX, startY, endX, endY;

This bit of nastiness fits an arc between the endpoints given.  If the
wo point are the same, then a loop is drawn.  Otherwise, if either of
the either of the coordinates are the same, a 180 degree arc is used
to  connect the points.  Otherwise, a 90 degree arc is made.

Note: For efficiency, shift-left and shift-right are used to multiply
and divide by two, just in case the optimizer doesn't do this
automatically.
```

### 5.13.2 DrawLink()

```
void
DrawLink(link, startX, startY, drawNow)
    LINK_PTR  link;
    int       startX, startY, drawNow;

Draw a link, given its starting coords.  If drawNow is TRUE, draw the
link on the screen immediately, as well as the pixmap, otherwise just
draw to the pixmap.
```

### 5.13.3 DrawAttLinks()

```
void
DrawAttLinks(inst, att, linkType, drawNow)
     INSTANCE_PTR  inst;
     ATTRIBUTE_PTR att;
     int           linkType,
                   drawNow;

Draw all links of an attributes links_list
```

### 5.13.4 DrawLinks()

```
void
DrawLinks(inst, linkType, drawNow)
     INSTANCE_PTR inst;
     int          linkType,
                  drawNow;

Draw all of an instance's links.
```

### 5.13.5 DrawAllLinks()

```
void
DrawAllLinks(drawNow)
     int drawNow;

Draw all links. If drawNow is TRUE, display them immediately, otherwise
just update the pixmap.
```

### 5.13.6 RestoreNoLinksDisplay()

```
void
RestoreNoLinksDisplay()

Description Copy the pixmap that does not show links to the pixmap to
  display.
Input, Output: None
```

# Chapter 6

# Objectbase Management

Marvel's objectbase manager is entirely contained within a module called obman.

## 6.1 Initialization

### 6.1.1 initialize_db()

```
int initialize_db(dbdir)
char *dbdir;
```

```
Initialize the database specified in the given string dbdir.  If it is
NULL, try to initialize the current directory.

Initialization includes changing to the proper directory and locking it.
```

### 6.1.2 lock_db()

```
static int lock_db(ob)
char *ob;
```

```
ob -- the objectbase in question.

Lock the objectbase by putting the string locked <person> in the
.marvel_rc file.  It is assumed that the file will be found in the
current directory, and has already been checked for writeability.
This routine is static to this file.

Unlock the database by removing the string from the file, hence
```

making it zero bytes.

### 6.1.3 unlock_db()

```
int unlock_db()
```

Unlock the database by removing and then touching the .marvel_id file
in the database directory.

## 6.2 Making Data Structures

### 6.2.1 make_attribute()

```
physadr make_attribute(name)
char *name;
```

Make an attribute. This is the non generic, low level routine. The
generic routine make_struct() should be used instead.

### 6.2.2 make_class()

```
physadr make_class(name)
char *name;
```

Make a class. This is the non generic, low level routine. The generic
routine make_struct() should be used instead.

### 6.2.3 make_graphic_info()

```
/*ARGSUSED*/
physadr make_graphic_info(name)
char *name;
```

Make a graphic_info. This is the non generic, low level routine.
The generic routine make_struct() should be used instead.

name is unused here.

### 6.2.4 make_instance()

```
physadr make_instance(name)
char *name;
```

Make an instance. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 6.2.5 make_obj_list()

```
/*ARGSUSED*/
physadr make_obj_list(name)
char *name;
```

Make an obj_list. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 6.2.6 make_obj_list_info()

```
/*ARGSUSED*/
physadr make_obj_list_info(name)
char *name;
```

Make an obj_list_info. This is the non generic, low level routine. The generic routine make_struct() should be used instead. Name is unused here.

### 6.2.7 make_struct()

```
physadr make_struct(type, name)
int type;
char *name;
```

This routine is intended to turn allocating of marvel structures more object oriented, in the sense of hiding details. NEVER use malloc to allocate and initialize structures, make_struct() does it all for you. ALWAYS use make_struct(), to avoid maximal debugging headaches.

type is defined in the appropriate include file, and is an integer which represents the structure in question.
name is the name of the primary name field of the record in question. For records which do not have such a field, just supply NULL. Name is copied, so the memory need not be persistent.

make_struct() returns a physadr, which is a guaranteed byte aligned pointer. The results should be cast in some appropriate fashion. This should ease some ofthe problems that might be encountered when going to a binary objectbase format.

In general, all pointers other then the main name (or similar field) are set to NULL. Numbers are usually set to -1, but that varies upon the structure.

These routines make one of something, for example, make_struct with a QUEUE_S flag passed will make an entry of a queue, RATHER THAN an entire queue.

### 6.2.8 make_subsuper()

```
physadr make_subsuper(name)
char *name;
```

Make an subsuper. This is the non generic, low level routine. The generic routine make_struct() should be used instead.

### 6.2.9 MA_free()

```
void MA_free(space)
char *space;
```

space -- the structure to be freed.

Free the space associated with string, which might have been a string, or other structure. When calling this routine, always cast

```
string to (char *), to keep lint happy.  Note that space should be
something that was allocated in one call, that is, you can not free a
structure with allocated character strings imbedded, you must first
seperately free the strings. All marvel routines should use this,
rather then directly calling free().
```

### 6.2.10 MA_malloc()

```
char *MA_malloc(size)
unsigned size;

   size - the amount of space to be allocated.

   Allocate the space desired.  The results should generally be cast
into some appropriate structure.  All marvel routines should use this
facility, rather then malloc, calloc, or some similar thing.
```

## 6.3 Objects

Following are all the middle level routines involved in making objects. Section 7.1 should also be referenced.

### 6.3.1 insert_attribute_instance()

```
int insert_attribute_instance(att, new_inst)
ATTRIBUTE_PTR att;
INSTANCE_PTR new_inst;


Insert an instance into a large att's list.  Be sure to check all
the cases.  if the instance in question is a top level instance do
nothing, that work is done by the complementary insert global routine.
if the inst in question is already on the list, do nothing and return
FALSE, otherwise return TRUE.

att is the owner attribute of the instance.  inst is the thang.
```

### 6.3.2 insert_global_instance()

```
int insert_global_instance(class, new_inst)
CLASS_PTR class;
INSTANCE_PTR new_inst;


Insert an instance into the global list.  Be sure to check all the
cases.  if the instance in question is a top level instance also fix
all the next and prev pointers to look the same and their global
counterparts.  This is done here to assure the pointers are the same,
and to avoid an extra loop through the objectbase.

This routine only fails of the inst in question is a top instance,
and a duplicate.

class is the owner class of the instance.  inst is the thang.
```

### 6.3.3 unlink_attribute_instance()

```
void unlink_attribute_instance(inst)
INSTANCE_PTR inst;


Unlink the prev and next pointers of an instance.  These are the ones
that connect attributes to instances to form hierarchy.
```

### 6.3.4 unlink_global_instance()

```
void unlink_global_instance(inst)
INSTANCE_PTR inst;


Unlink the global prev and global next pointers of an instance.
```

## 6.4 Attributes

Following are all the middle level routines involved in properly creating attributes. Section 6.6 is also relevent.

### 6.4.1 copy_all_small_atts()

```
void copy_all_small_atts(object, orig_obj)
INSTANCE_PTR object, orig_obj;
```

Copy all the small attributes of the template class of an object.
This routine does all the proper linking and such.

If orig_obj is not NULL, then use it's small attriutes to get values for what is being copied.

### 6.4.2 copy_large_att()

```
ATTRIBUTE_PTR copy_large_att(orig_att)
ATTRIBUTE_PTR orig_att;
```

Make a copy of the given large attribute.

### 6.4.3 copy_med_att()

```
ATTRIBUTE_PTR copy_med_att(orig_att)
ATTRIBUTE_PTR orig_att;
```

Make a copy of the given medium attribute.

### 6.4.4 copy_small_att()

```
ATTRIBUTE_PTR copy_small_att(orig_att)
ATTRIBUTE_PTR orig_att;
```

Make an attribute as a copy of a given template, usually from a class. The attribute is copied identically, unless it is an auto initable kind of attribute, in which case it gets current values. Note that this means that if an object is used as the template, user and time type of attributes will not be the same, other routines would have to be used for this. See copy_or_find_inst_att() for more details.

## 6.5 Inheritance

### 6.5.1 check_class_inheritance()

```
int check_class_inheritance(class1, class2)
CLASS_PTR class1, class2;
```

This routine checks to see if class2 is a superclass of class1, or the same class. If so, it returns TRUE, otherwise it returns FALSE.

### 6.5.2 check_ss_class_inheritance()

```
int check_ss_class_inheritance(class1, class2)
CLASS_PTR class1, class2;
```

This routine checks to see if class2 is either a superclass or subclass of class1, or the same class. If so, it returns TRUE, otherwise it returns FALSE.

### 6.5.3 check_sub()

```
static int check_sub(sub, class)
SUBSUPER_PTR sub;
CLASS_PTR class;
```

Check to see of the given class is in the given list of subclasses. This must be done recursively, so for each subclass, all that classes subclasses must be checked.

True is returned if the class is in the subclass list given, otherwise FALSE.

### 6.5.4 check_super()

```
static int check_super(super, class)
SUBSUPER_PTR super;
```

```
CLASS_PTR class;
```

Check to see of the given class is in the given list of superclasses. This must be done recursively, so for each superclass, all that classes superclasses must be checked.

True is returned if the class is in the subclass list given, otherwise FALSE.

### 6.5.5 get_inherited_atts()

```
void get_inherited_atts(class, object)
CLASS_PTR class;
INSTANCE_PTR object;
```

```
class -- the class to check for superclasses.  If inheritance is
         desired with the objects owner, class can be passes as NULL.
object -- The object recieve all the inherited attributes.
```

Get all the attributes that are inherited from other object classes. This is a recursive procedure, if class is null, then start at the begining, else start looking at the class class.

We follow the following rules:
1. use local definition of attribute if found.
2. use closest definition as we go up the hierarchy. This means take the first superclass on the list, search all the way up its inheritance hierarchy, take the next one, and so on, recursively.

This routine currently only gets the small inherited atts and the large, link type inherited atts.

Be certain to ignore superclasses with no address, these are things like "TOOL" and "ENTITY".

### 6.5.6 get_large_inherited_atts()

```
void get_large_inherited_atts(class, object)
CLASS_PTR class;
INSTANCE_PTR object;
```

```
Get all the large link type inherited atts from the given class, and
put copies of them into the object.
```

### 6.5.7 get_med_inherited_atts()

```
void get_med_inherited_atts(class, object)
CLASS_PTR class;
INSTANCE_PTR object;

Get all the medium link type inherited atts from the given class, and
put copies of them into the object.
```

### 6.5.8 get_small_inherited_atts()

```
void get_small_inherited_atts(class, object)
CLASS_PTR class;
INSTANCE_PTR object;

Get all the small link type inherited atts from the given class, and
put copies of them into the object.
```

Functions to find inherited attributes are detailed in Section 6.6

## 6.6 Traversing the Objectbase

Classes

### 6.6.1 find_class()

```
CLASS_PTR find_class(name)
char *name;

Find a class from the systems list of classes, given it's name.  Returns
NULL upon failure.
```

### 6.6.2 find_class_given_root()

```
CLASS_PTR find_class_given_root(root, name)
CLASS_PTR root;
char *name;
```

Find a class from a list of passed classes, given it's name. Returns NULL upon failure.

Objects

### 6.6.3 find_first_obj_of_class()

```
INSTANCE_PTR find_first_obj_of_class(class, oname)
CLASS_PTR class;
char *oname;
```

class -- a pointer to a class
oname -- the name of the object in question.
object -- the object to look for the last occurance of

Find the first and last objects on the class's global object list. The last routine should only be called after the first one, if you don't want them to blow up. There is no particular way of verifying it's input so it should be used carefully. These routines do not use the inheritance mechanism.

### 6.6.4 find_last_obj_of_class()

```
INSTANCE_PTR find_last_obj_of_class(object)
INSTANCE_PTR object;
```

Find the last object the object's class's global list with the same name.

### 6.6.5 find_obj()

```
INSTANCE_PTR find_obj(oclass, oname, start_from_root)
CLASS_PTR oclass;
char *oname;
int start_from_root;


oclass -- a pointer to a class specified by the rule.
oname -- the name of the object in question.
start_from_root -- TRUE to start the search from the root,
                   FALSE otherwise.
```

find_obj() is the general way of finding an object in the objectbase, using the inheritance mechanism.

Input is a simple name of an object <oname> (read object id), a class <oclass> that the object must belong (that class can be a superclass, of course), and a flag that specifies whether to do the local search or not. The routine uses the concept of current object otherwise, which is always set. A pointer to the object in question is returned if only one unique object was found, otherwise, a list of objects is printed. Here is the algorithm:

1. If the current object's name is <oname>, and its class is <oclass> then this is the one we want. Return it. Otherwise go on.

2. Search for objects of type <oclass>, or subclasses of <oclass>, amongst the current object's progeny, recursively. If lists of such objects are found, search each found list for an object named <oname>. There will be at most one such object in each of these lists. If the end result is only one object, return it. Otherwise, if many are found, print all the possibilities, and return NULL. Otherwise go on. This step goes on for each distinct hierarchical level in the objectbase below the current object, which higher hierarchical levels getting priority. Note that we have an odd way of treating inheritance precedence here, it is ignored in favor of object hierarchy precedence. I think this is the right way to do it, as an example should convince.

3. If 1. and 2. fail, go to the root of the objectbase. All objects belonging to class <oclass>, or having <oclass> as a superclass (recursively) must be searched in this case. If this search yields one object only, return it. Otherwise, print out all the duplicate objects and return NULL.

### 6.6.6 find_obj_downwards()

```
INSTANCE_PTR find_obj_downwards(start_obj,oclass,oname,pdist,level_diff,
                                start_from_root)
INSTANCE_PTR start_obj;
CLASS_PTR oclass;
char *oname;
int *pdist,level_diff;
int start_from_root;


oclass -- a pointer to a class specified by the rule.
oname -- the name of the object in question.
pdist -- an output variable , will hold the distance from curr_obj.
level_diff -- the level difference. each level increments this
              variable by 200. notice that we assume that this is
              max number of objects in level (i.e subtree).
              if larger objectbases are expected this constant should
              be incremented.
              when pdist is updated whithin its level, the prefix
              of level_diff is added.
start_from_root -- TRUE to start the search from the root,
                   FALSE otherwise.

This function scanns the subtree of start_obj, for object named
oname that is of type of subclass of oclass.  returns the object and
its pdist if found otherwise NULL and CLASS_NOT_FOUND

Input is a simple name of an object <oname> (read object id), a
class <oclass> that the object must belong (that class can be a
superclass, of course),a pointer to the distance of the found object
from the curr_obj, and a flag that specifies whether to do the local
search or not.  The routine uses the concept of current object
otherwise, which is always set.  A pointer to the object in question
is returned if only one unique object was found, otherwise, a list of
objects is printed.  The algorithm is almost simmilar to find_obj,
except :

    1. it starts from start_obj, an input parameter.
    2. it has two modes:
       a. ! start_from_root: - does step 1 and step 2 as in find_obj.
       b. start_from_root: - does step 3 of find_obj.  NOTE:
          single_element_object_list_with_level is called here,
          instead of single_element_object_list_at_level. The new
          function gets the distance of obj from curr_obj in
          addition to the object, as returned values.
```

### 6.6.7 find_obj_of_att()

```
INSTANCE_PTR find_obj_of_att(att, name)
ATTRIBUTE_PTR att;
char *name;


att -- a pointer to the attribute to start the search.  It should be
       a set of type of attribute, for now.
name -- the object name (oid) to look for.

Find an object, given an attribute of another object.  This routine
just searches the attribute's instance list.  Note that there can not
be multiple instances of the same name here, as with the similar class
finding routine.  The inheritance mechanism is not used in this
routine.
```

### 6.6.8 find_obj_of_path()

```
INSTANCE_PTR find_obj_of_path(path)
char *path;


path -- the path, expected to be from the current instance, to
        search for.

Unlike any of the other object finders, this one starts from the
current inst, and looks for the inst based upon the elements in the
path given.  Thus, this search is hierarchical. Just traverse all
possible paths of the hierarchy, and return the instance.

Paths are of the form obj_name/att_name/obj_name...
A local path will always start with an attribute name,
one with a / will start with an object name.  Note that all paths
are guaranteed to be unique in this fashion, because each attribute
can only appear once in an object, and parent attrbitutes can't
have multiple children with the same name.

A . is acceptable, the current inst is returned.  A / causes the
search to start from the root.  Other special characters are
meaningless.

This routine is mostly used to find objects in the command line
interface.
```

### 6.6.9 find_obj_with_dist()

```
INSTANCE_PTR find_obj_with_dist(oclass, oname, pdist,
                                start_from_root)
CLASS_PTR oclass;
char *oname;
int *pdist;
int start_from_root;


oclass -- a pointer to a class specified by the rule.
oname -- the name of the object in question.
pdist -- an output variable , will hold the distance from curr_obj.
start_from_root -- TRUE to start the search from the root,
                   FALSE otherwise.
```

find_obj_with_dist() is the DWIM way of finding an object in the objectbase, using the inheritance mechanism.

Input is a simple name of an object <oname> (read object id), a class <oclass> that the object must belong (that class can be a superclass, of course),a pointer to the distance of the found object from the curr_obj, and a flag that specifies whether to do the local search or not. The routine uses the concept of current object otherwise, which is always set. A pointer to the object in question is returned if only one unique object was found, otherwise, a list of objects is printed. Here is the algorithm: (we'll refer to currently scanned object as OBJ to distinguish from the current_object in the objectbase)

```
1.  while parent of OBJ is not NULL (i.e OBJ is root of tree)
    2.  mark OBJ as visited
    3.  look for obj in OBJ sons'  subtrees excluding those marked
        visited. (using find_obj_downwards)
    4.  if found -
            return it.
    5.  else
         6.  OBJ <-- OBJ's parent
         7.  update distance
    end of loop
8.  iterate once more for the root.
9.  now scan the rest of the forest, by applying find_obj_downwards
    to each of the (unvisited) roots.
```

### 6.6.10 find_progeny_object()

```
static INSTANCE_PTR find_progeny_object(obj, oname)
INSTANCE_PTR obj;
char *oname;
```

```
obj -- the head of the list of objects to be searched.
oname -- the object name (oid) to search for.

Use the next field, not the global next, of a list of objects, to
search for an object.  This routine is static.
```

Attributes

### 6.6.11 copy_or_find_inst_att()

```
ATTRIBUTE_PTR copy_or_find_inst_att(inst, att)
INSTANCE_PTR inst;
ATTRIBUTE_PTR att;
```

```
inst -- the instance in which you are searching for the attribute.
att -- the list to use to look for templates.

search for an instance's attribute with the same name as the passed
attribute, and if found, return the instance's copy.  If not found,
add one to the instance's list, and copy all the goodies from the
passed attribute.

The passed attribute might either be hanging off a class or another
instance.  In either case, we want the exact attribute.  But this is
slightly tricky, the class attributes might need to have some things
autoinited, and the other ones just want copies.  copy_small_att()
takes care of this.
```

### 6.6.12 find_class_small_att()

```
ATTRIBUTE_PTR find_class_small_att(class, name)
CLASS_PTR class;
char *name;
```

Find the small attribute specified by the given name in the class given. First look at that class, then use inheritence to look at all the superclasses of the passed class by calling find_inherited_small_att(). The find is done by a string compare of the name of the attribute. Return the found template attribute, or NULL if none is found.

There are complementary routines for medium and large attributes called find_class_med_att() and find_class_large_att().

### 6.6.13 find_class_med_att()

```
ATTRIBUTE_PTR find_class_med_att(class, name)
CLASS_PTR class;
char *name;
```

Find the medium attribute specified by the given name in the class given. First look at that class, then use inheritence to look at all the superclasses of the passed class by calling find_inherited_small_att(). The find is done by a string compare of the name of the attribute. Return the found template attribute, or NULL if none is found.

### 6.6.14 find_class_large_att()

```
ATTRIBUTE_PTR find_class_large_att(class, name)
CLASS_PTR class;
char *name;
```

Find the large attribute specified by the given name in the class given. First look at that class, then use inheritence to look at all the superclasses of the passed class by calling find_inherited_small_att(). The find is done by a string compare of the name of the attribute. Return the found template attribute, or NULL if none is found.

### 6.6.15 find_instance_attribute()

```
ATTRIBUTE_PTR find_instance_attribute(object, name)
INSTANCE_PTR object;
char *name;
```

Find the attribute corresponding to the given name from the given object. First large, then medium, then small ones are searched. Technically, I believe that between the different kinds of attributes there can be duplicate names, this routine does not handle that.

### 6.6.16 find_object_large_att()

```
ATTRIBUTE_PTR find_object_large_att(object, name)
INSTANCE_PTR object;
char *name;
```

Find the large attribute whose name matches the given name. Use the given object's list of large attributes to search from.

### 6.6.17 find_object_med_att()

```
ATTRIBUTE_PTR find_object_med_att(object, name)
INSTANCE_PTR object;
char *name;
```

Find the medium attribute whose name matches the given name. Use the given object's list of medium attributes to search from.

### 6.6.18 find_object_small_att()

```
ATTRIBUTE_PTR find_object_small_att(object, name)
INSTANCE_PTR object;
char *name;
```

search the object given, and find the specified attribute. Do not check for inheritance, because we are looking for actual attributes here, and inherited small and medium ones will be already present. Large ones are there is so added by addinst.

find_instance_attribute first looks for large, then medium, then small ones.

object -- the object in which to search for the attribute.
name -- the name of the attribute.

### 6.6.19 find_inherited_large_att()

```
ATTRIBUTE_PTR find_inherited_large_att(class, name)
CLASS_PTR class;
char *name;
```

Find the large attribute whose name is the given name, from the class given. Use the inheritance mechanism if need be.

### 6.6.20 find_inherited_med_att()

```
ATTRIBUTE_PTR find_inherited_med_att(class, name)
CLASS_PTR class;
char *name;
```

Find the medium attribute whose name is the given name, from the class given. Use the inheritance mechanism if need be.

### 6.6.21 find_inherited_small_att()

```
ATTRIBUTE_PTR find_inherited_small_att(class, name)
CLASS_PTR class;
char *name;
```

Find the small attribute whose name is the given name, from the class given. Use the inheritance mechanism if need be.

### 6.6.22 is_class_or_subclass()

```
int is_class_or_subclass(c1, c2)
CLASS_PTR c1,c2;


c1 -- a pointer to a class.
c2 -- a pointer to another class, which will be checked for
      being a subclass.

Returns TRUE if the c2 is either the same class as c1, or a
subclass (recursively) of c1.  Find this out by checking c1's
subclasses (not c2's!).
```

### 6.6.23 make_progeny_object_list()

```
static void make_progeny_object_list(class, start_obj, oname, level)
CLASS_PTR class;
INSTANCE_PTR start_obj;
char *oname;
int level;


class --  The class to check for class inheritance.
start_obj -- where to start the search.
oname -- the name of the object to search for.
level -- the level away from the root.  Starts with 1.

Make a list of progeny objects, starting with some root, which
might be a top level object, and might be the current object.
```

See also the function find_link() in Section 6.11.

## 6.7 Objectbase Variables

### 6.7.1 get_cur_instance()

```
INSTANCE_PTR get_cur_instance()


Get the current object.  Note that this might be NULL.
```

### 6.7.2 get_db_name()

```
char *get_db_name()
```

Get the name of the objectbase.

### 6.7.3 get_db_root()

```
CLASS_PTR get_db_root()
```

Get the objectbase root.

### 6.7.4 get_marvel_pid()

```
int get_marvel_pid()
```

Get the current marvel process id.

### 6.7.5 set_cur_instance()

```
void set_cur_instance(new_obj)
INSTANCE_PTR new_obj;
```

Set the current object to some new object. Avoid setting it to NULL unless the objectbase is truely empty.

### 6.7.6 set_db_name()

```
void set_db_name()
```

This odd one sets the dbname to be the current working directory. It is only used in init, but get_db_name, which gets the name back, is used in a variety of places.

### 6.7.7 set_db_root()

```
void set_db_root(head)
CLASS_PTR head;
```

Set the root of the objectbase. Done in init time

### 6.7.8 set_marvel_pid()

```
void set_marvel_pid()
```

Set the pid of the marvel process, so this sys call need not be repeated all the time.

## 6.8 Making Files and Directories

### 6.8.1 exists_dir()

```
int exists_dir(name, mode)
char *name;
int mode;
```

name -- the name of the file or directory
mode -- the permissions tor check for, as below.

These routines check accessibility of files and directories, respectively. They do not guarantee writeablility, but indicate as much certainty as can be had.

The modes are:

| | |
|---|---|
| READ | read perms |
| READEXECUTE | read/execute perms |
| READWRITE | read/write perms |
| READWRITEEXECUTE | read/write/execute perms |

TRUE is returned if accessibility is there, FALSE otherwise.

### 6.8.2 exists_file()

```
int exists_file(name, mode)
char *name;
int mode;
```

name -- the name of the file or directory
mode -- the permissions tor check for, as below.

These routines check accessibility of files and directories, respectively. They do not guarantee writeablility, but indicate as much certainty as can be had.

The modes are:

| | |
|---|---|
| READ | read perms |
| READEXECUTE | read/execute perms |
| READWRITE | read/write perms |
| READWRITEEXECUTE | read/write/execute perms |

TRUE is returned if accessibility is there, FALSE otherwise.

### 6.8.3 make_directory()

```
int make_directory(newdir)
char *newdir;
```

This routine makes a directory in the path given. The path to the new directory name must exist.

### 6.8.4 make_inst_disk_structures()

```
int make_inst_disk_structures(inst)
INSTANCE_PTR inst;
```

Make disk structures corresponding to this object.

## 6.9 Object Lists

### 6.9.1 add_to_obj_list()

```
void add_to_obj_list(manager, obj, level)
OBJ_LIST_INFO_PTR manager;
INSTANCE_PTR obj;
int level;
```

Add an object to an objectlist.

### 6.9.2 clear_obj_list()

```
void clear_obj_list(manager)
OBJ_LIST_INFO_PTR manager;
```

Clear the object list pointed to by the given manager.

### 6.9.3 empty_obj_list()

```
int empty_obj_list(manager)
OBJ_LIST_INFO_PTR manager;
```

TRUE if the given object list is empty. FALSE otherwise.

### 6.9.4 empty_obj_list_at_level()

```
int empty_obj_list_at_level(manager, level)
OBJ_LIST_INFO_PTR manager;
int level;
```

TRUE if the given object list is empty at a specified level, FALSE otherwise.

### 6.9.5 get_next_in_list()

```
INSTANCE_PTR get_next_in_list(manager)
OBJ_LIST_INFO_PTR manager;
```

### 6.9.6 get_next_obj_from_list()

```
INSTANCE_PTR get_next_obj_from_list(manager, level)
OBJ_LIST_INFO_PTR manager;
int level;
```

Gets the head object from the object list, and points the manager at the next one. Does not remove the object from the list.

### 6.9.7 init_obj_list()

```
OBJ_LIST_INFO_PTR init_obj_list()
```

Initialize an object list, returning a pointer to a new manager.

This object list manipulation package also includes:

```
   void clear_obj_list(manager)
   int empty_obj_list(manager)
   empty_obj_list_at_level(manager, level)
   single_element_obj_list(manager)
   single_element_obj_list_at_level(manager, level)
   print_obj_list(manager)
   add_to_obj_list(manager, obj, level)
   get_next_obj_from_list(manager, level)
   pop_first_obj_from_list(manager, level)
   reset_next_in_list(manager);
   get_next_in_list(manager);

   INSTANCE_PTR obj -- a pointer to an object to add to the list
   int level -- a user set level, to divide lists into sublists. Level
                must be >= 0 for the package to operate correctly.
   OBJ_LIST_INFO_PTR -- manager;
```

```
This set of routines provides a seperate manipulation package for
Marvel to have lists of objects found at a given time.  It is used by
various find routines, but anyone is welcome to use them.  The macro
IGNORE_LEVEL should be used in routines where level is not a concern.

Users must initialize and clear all the object lists that they use.

The list stores objects and levels, levels are user set.  All the
pointers to the list are static to this file, so they cannot be
accessed except through this package.

clear_obj_list() frees memory, and initializing the package.  It
should always be used before using any of the other functions.

empty_obj_list() tells if the current obj_list if empty or not.
TRUE is returned if empty, FALSE otherwise.

empty_obj_list_at_level() tells if the current obj_list if has any
elements of the level given or not.  TRUE is returned if empty, FALSE
otherwise.

single_element_obj_list() tells if the current obj_list has only
one entry, and returns it if this is the case.  An object is returned,
not the list.

single_element_obj_list_at_level tells if the current list has only
one entry at the specified level.  If level is IGNORE_LEVEL, then a
single element of the lowest level, starting from 0, is searched for.
An object is returned if uniquely found.

print_obj_list() prints out the list.  If level is not
IGNORE_LEVEL, only those at the given level are printed.

add_to_obj_list() adds an object to the list.  Level and an object
pointer must be specified.

get_next_obj_from_list() gets the next object from the list.  If a
level is specified, the next one is only advanced and returned if it
is of that level.  pop_first_obj_from_list() pops the first object
from the list.  Level is thus inconsequential.

reset_next_in_list(manager) resets the next pointer in the manager
to the begining of the list.

get_next_in_list(manager) gets the next one in the list, and
advances the next pointer.
```

### 6.9.8 pop_first_obj_from_list()

```
INSTANCE_PTR pop_first_obj_from_list(manager)
OBJ_LIST_INFO_PTR manager;
```

Returns and removes the first object from the list.

### 6.9.9 print_obj_list()

```
void print_obj_list(manager, level)
OBJ_LIST_INFO_PTR manager;
```

Prints the names of all the objects on the object list.

### 6.9.10 reset_next_in_list()

```
int reset_next_in_list(manager)
OBJ_LIST_INFO_PTR manager;
```

Resets the next object in the list to be the first one again.

### 6.9.11 single_element_obj_list()

```
INSTANCE_PTR single_element_obj_list(manager)
OBJ_LIST_INFO_PTR manager;
```

Returns the object if it is the only one in the list, otherwise NULL.

### 6.9.12 single_element_obj_list_at_level()

```
INSTANCE_PTR single_element_obj_list_at_level(manager, level)
OBJ_LIST_INFO_PTR manager;
int level;
```

```
Returns the object if it is the only one in the list at the
given level, otherwise NULL.
```

### 6.9.13 single_element_obj_list_with_level()

```
INSTANCE_PTR single_element_obj_list_with_level(manager,plevel)
OBJ_LIST_INFO_PTR manager;
int *plevel;
```

### 6.9.14 derive_inst_path()

```
char *derive_inst_path(obj)
INSTANCE_PTR obj;
```

```
obj -- the object whose path is desired.

Follow the objectbase hierarchy upwards, and get a path like  string
which corresponds to a directory structure of where this instance
can be found.  Find_path() tacks on the path to the dbroot, for use
with envelopes, while derive_inst_path() just gives the internal path.

In both cases, a static char * is returned.
```

### 6.9.15 find_path()

```
char *find_path(obj)
INSTANCE_PTR obj;
```

```
obj -- the object whose path is desired.

Follow the objectbase hierarchy upwards, and get a path like  string
which corresponds to a directory structure of where this instance
can be found.  Find_path() tacks on the path to the dbroot, for use
with envelopes, while derive_inst_path() just gives the internal path.

Also find_path() puts in the names of large attributes in between
```

```
all the objects.

In both cases, a static char * is returned.
```

## 6.10 Generic Lists

These functions are used to create generic linked lists. The nodes consist of two pointers: a pointer to a physadr (that is any item), and a pointer to the next node. They are used in the link module to build up menus (Palette Items) but they could be used in any part of Marvel (remember to make them external functions).

### 6.10.1 CreateListNode()

```
static LIST_PTR
CreateListNode(struct_ptr)
     physadr struct_ptr;


Description: Create one node of a list of generic pointers, and return
pointer to that node.  Note: this function is declared as static, but,
if other modules have a use for the generic list structures, the
programmers should feel free to change that to external.

input:  struct_ptr -- pointer to which the list node will point

output: pointer to new generic list node.
```

### 6.10.2 FreeListNodes()

```
void
static FreeListNodes(list)
     LIST_PTR list;


Description: Free all nodes in the given list.  The list is a link list
   of the geric list type LIST_PTR which is described in dbman.h.

Input:
  list -- a pointer to the head of the list of nodes to be freed.
```

### 6.10.3 ListLength()

```
static int
ListLength(lp)
     LIST_PTR lp;


  Description: Count and return the number of nodes in an list.

  input: lp -- pointer to the head of the generic link list to count.
  output: the number of nodes in that list , 0 => (lp == NULL).
```

### 6.10.4 ListNth()

```
static LIST_PTR
ListNth(lp, n)
     LIST_PTR lp;
     int n;


  Description: Return a pointer to the nth node of a generic linked list.

  input: lp -- pointer to the head of the list to search
         n  -- the number of the node to find (the n in nth).
 n is not from 0, like arrays, ie. the first node is the 1st node,
 not the 0th node.  If lp is input as 0, NULL is returned.
  output: see description!
```

## 6.11 Graphical Links

**Introduction**

The graphical links module supports the creation of knowledge base webs. Using graphical links (also known as "links") Marvel applications can represent and use directed or undirected graphs in addition to the Marvel organizational hierarchy.

Each link is described as a tuple: (Source,Destination) where Source is an attribute of type "link" and Destination is either an attribute or an instance. Multiple links may originate from one link attribute, and a single instance or attribute may be the destination for multiple links. The number of total links is limited only by the available dynamic memory and disk space. Cyclic, including circular links, are permitted.

**The Links Data Structure**

From each source attribute there is a list of link nodes to which the member of the attribute structure, "links_list" points. Non-link attributes and link attributes which are not pointing to anything have their "links_list" member set to NULL.

The link nodes consist of 5 members. The "tag" member is an unique id number, which is assigned to each link upon its creation. "Item" is a pointer to the destination object, the "linkType" field indicates whether "item" points to an attribute or an instance. Finally, "next" points to the next link in the list, i.e. the next link whose source is the same as the link in question. In Figure 6.1, the internal reprensentation of the graphical links is shown. There are two links, one in which the destination is an instance (tag = 7), and one which points to an attribute (tag = 15).

Each destination object has a member "own_link_list" which points to a link list of owner link nodes (also known as back link nodes). These structures conatin three fields: "own_link_ptr" which points to the link node in question, "tag" which is just the tag of the link in question, and "next" which points to the next owner link node for this destination.

Link Creation and Deletion Functions

### 6.11.1 add_link()

```
LINK_PTR
add_link(link_attribute, destination_instance,
  destination_attribute,link_tag)
     ATTRIBUTE_PTR link_attribute, destination_attribute;
     INSTANCE_PTR  destination_instance;
     int link_tag;


Input:

link_attribute        -- the link attribute to attach given instance

<one of the following must be NULL>
destination_instance  -- the instance to attach.
destination_attribute -- the instance to attach.

Output:

RETURN  -- a pointer to the link created, null if none could be created
because such a link already existed.

Description:

Add destination_instance/destination_attribute to the
link_attribute's LINK list.
```

### 6.11.2 GetLinkTag()

```
static int GetLinkTag()
```

Figure 6.1: Graphical Link Datastructure.

```
This function returns the number to be used for a new-link's id-tag.
It increments MaxLinkTag, and then returns that new number.

Input: None
Output: the (incremented) MaxLinkTag
```

### 6.11.3 add_link_to_tag_list()

```
void
  add_link_to_tag_list(new_link_tag, new_link, own_link_list_ptr)
int new_link_tag;
LINK_PTR new_link;
OWN_LINK_PTR *own_link_list_ptr;
```

### 6.11.4 delete_link()

```
static int
delete_link(link_attribute, item_to_remove)
     ATTRIBUTE_PTR link_attribute;
     physadr       item_to_remove;


Input:

link_attribute  -- the link attribute to remove given instance from
item_to_remove  -- the instance or attribute to remove.

Output:

RETURN  -- FALSE if attribute is not of LINK_ATT, else TRUE

Description:

Remove instance_to_remove from the link_attribute's LINK list, if the
link is found and removed, return TRUE. If the link is not found,
and therefore -- obviously -- not removed, return FALSE. Also,
find the back-link node (at the destination object) which points
to this link, and delete it.  Note that this is the function which
deletes links from "in front", i.e. when unlink_cmd is called, as
opposed to delete_link_from_behind.
```

When deleting a Marvel instance, any links that point to the object to be deleted, as well as any links whose source is an attribute of the object to be deleted, must be removed. This process is accomplished by following the owner link nodes for the instance

and for all of its attributes back to the links which point to them. The owner_attribute pointers of the links are then traversed to find the source of the link. Then, the link is excised from the links list of its source, and the link node and the owner link node are removed. All link nodes which originate in one of the link attributes of the object to be deleted are removed using delete_link.

### 6.11.5 remove_links_to_from_inst()

```
void remove_links_to_from_inst(inst)
     INSTANCE_PTR inst;


Description:
This function deletes all of the links which point to an instance or
any of the instance's attributes.  It is used when deleting an instance
to free up the links for which the instance (or an attribute of the
instance) is the destination.

1/26/90: The links which eminate from the instance are also removed.

Input: inst -- the instance in question (i.e. the instance which will
       be deleted)
Output: none

The function is called externally (see delete.c)
```

### 6.11.6 delete_link_from_behind()

```
static void
  delete_link_from_behind(link_to_del)
LINK_PTR link_to_del;


description: this function is used to delete a link given only a pointer
to that link.  Unlike delete_link, it does not worry about removing the
back link pointer which points to this link at the destination object.
This is because the link pointer was originally located from the
back link at the dest. obj. (see remove_back_links).

input: a pointer to the link to delete.
Output: none.
```

### 6.11.7 remove_back_links()

```
static void
  remove_back_links(own_list_ptr)
OWN_LINK_PTR *own_list_ptr;


description: this function is used to remove the links which are
pointed to by the own_list link list passed in.  It is used for deleting
links when starting from the destination object.  The function simply
iterates through all of the back-link nodes -- deleting the link and
then freeing the node.

input: pointer to the head of the back-links list (the owner list).
Output: None.
```

### 6.11.8 remove_link_from_list()

```
static void
  remove_link_from_list(link_to_delete)
LINK_PTR link_to_delete;

input: link_to_delete -- pointer to a link structure which indicates the
link who's back-link node should be deleted.

output: none

description: the function is used when deleting a link.  It searches the
own_link_list of the destination object of the link for the back_link
node which points to the link in question.  Once found, it deletes it
and frees the space.  The free-ing of the link structure itself is left
to the calling program (delete_link).
```

#### Querying the Status of Links

The following functions are used by the X-interface to query the status of one or more links.

### 6.11.9 GetLinks()

```
LIST_PTR
GetLinks(att, type_spec)
     ATTRIBUTE_PTR att;
     int type_spec;
```

```
builds up a list of all of the links in att which are of the
specified link type (i.e. anAttribute or anInstance). If type_spec is
-1 (ALL_LINK_TYPES), all links are used. If inst is null, or if there are
no links of the specified type, NULL is returned.

att       -- the attribute in which to search for the links
type_spec -- the type of the links which should be included. If
             the type is ALL_LINK_TYPES (-1), then all types are included.
```

### 6.11.10 ActuallyLinked()

```
int
ActuallyLinked(sourceAtt, destInst, linkType)
     ATTRIBUTE_PTR sourceAtt;
     INSTANCE_PTR  destInst;
     int           linkType;

description: this function returns TRUE iff the destination instance is
linked to the source attribute

output: TRUE/FALSE
```

A more general function is provided below for checking the status of a particular link between two known objects. This function is not used by the link module internally, but is provided so that other modules of marvel (e.g. the evaluator) may test the state of the links.

### 6.11.11 find_link()

```
LINK_PTR
  find_link(sourceAtt, destobj, search_code)
ATTRIBUTE_PTR  sourceAtt;
physadr        destobj;
int            search_code;

NOTE: At present, this function is un-used  -- for evaluator.

Given the source attribute of a link and proposed destination instance
the function returns a pointer to the link  which connects the
sourceAtt-ribute to the inst.

Input:
sourceAtt -- the source attribute for the specified link
```

```
Note: one of the following two args should be == NULL
destobj  -- the proposed destination of the link we're trying to find.

search_code --  0 ==> search for a link to destobj; destobj
                      may be an instance or an attribute.
                1 ==> search for a link to destobj; destobj must
       be an instance.
                2 ==> search for a link to destobj; destobj must be
       an attribute.
                3 ==> destobj must be an instance: search for a link
       to any of destobj's attributes.
       -->first link found is returned.

Output: A pointer to the found link, or NULL if none was found.
```

### X Interface Functions for Links

When linking or unlinking while using the X-interface, the user selects the source and destination objects by pointing to instances and choosing appropriate attributes from menus. The source instance, i.e. the owner instance of the source attribute, is always assumed to be the current instance.

## 6.11.12 GetInstanceFromUser()

```
INSTANCE_PTR
GetInstanceFromUser()


  Get and return an instance selection from the user (X interface).
```

## 6.11.13 PickDestInstance()

```
static INSTANCE_PTR
PickDestInstance(sourceInst, sourceAtt, linkType)
     INSTANCE_PTR  sourceInst;
     ATTRIBUTE_PTR sourceAtt;
     int           linkType;

description:
Given a source instance/attribute, get a user pick to be used as the
destination instance for the link.  If the instance picked is not
already linked to, then return NULL (fail).
```

## 6.11.14 PickAttribute()

```
ATTRIBUTE_PTR
PickAttribute(inst, type_spec, className)
     INSTANCE_PTR    inst;
     int type_spec;
     char *className;


  Put up a scrolling menu of all attributes of the given instance which
  conform to the type_spec, and get the user's selection of one of those
  attributes.  For info on the type_spec and single_flag params, see
  GetAttributes -- these params are just passed along to that function.
```

### 6.11.15 PickDestAttribute()

```
static ATTRIBUTE_PTR
PickDestAttribute(links, inst)
     LINK_PTR        links;
     INSTANCE_PTR    inst;

description:  this function allows the user (in the XFACE) to pick an
attribute which is pointed to by one of the links in a list of links.

input:  links -- the pointer to the head of a list of links
        inst  -- the instance who's attributes are to be searched.

output:  pointer to the attribute selected.
```

### 6.11.16 GetAttributes()

```
static LIST_PTR
GetAttributes(inst, type_spec,className)
     INSTANCE_PTR inst;
     int type_spec;
     char *className;


  Description:
  Build up a list of all of the attributes in inst which are of the specified
  type.   If inst is NULL, or if there are no attributes of the specified
  type, NULL is returned.

  Input:
  inst      -- the object in which to search for the attributes
```

```
  type_spec -- the type of the attributes which should be included.

  Output: See description!
```

### 6.11.17 GetLinkedAttributes()

```
LIST_PTR
GetLinkedAttributes(links, inst)
     LINK_PTR        links;
     INSTANCE_PTR    inst;

description:  returns a list of all links in the given links list which
point to an attribute in the given instance

input: links -- a pointer to the list of links in question.
       inst  -- the given instance

output: a generic list which points to the relevant links.
```

### 6.11.18 GetLinkListTypes()

```
void
GetLinkListTypes(links, insts, atts)
     LINK_PTR  links;
     int       *insts, *atts;

description: this function sets flags corresponding to whether there
are any links to instances and/or links to attributes in the given
"link" list.

input: links -- pointer to the head of a list of links
       insts -- pointer to a flag which will be set to TRUE if any of the
                links in "links" points to an instance
       atts  -- same thang as insts except for attributes.

output: none
```

### 6.11.19 MakeAttArray()

```
static PaletteItem*
MakeAttArray(lp,length)
LIST_PTR lp;
```

```
description: builds an array of PaletteItems given a generic linked list
in which "da_strcut_ptr" points to an ATTRIBUTE structure.  The name of
each of these attributes is used to create a menu in which each choice is
the name of an attribute.

input: lp     -- pointer to the head of the generic linked list.
       length -- num of items in list.  If it is <= 0, the func will
         calculate it automatically.

 output: a pointer to the first address of the PaletteItem array.
```

### 6.11.20 MakeLinkArray()

```
static PaletteItem*
MakeLinkArray(lp,length)
LIST_PTR lp;


description: This function builds up an array of PaletteItems given a
list

lp -- pointer to the head of the list.
length -- number of items in list.  If length is <= 0, the function
will calculate the length.
```

### 6.11.21 MakeMenu()

```
static ControlPtr
MakeMenu(items, numOfItems)
     PaletteItem items[];
     int         numOfItems;


  Return a pointer to a scrollable menu created using the information supplied
  in the items parameter.
```

#### Saving and Loading Links from the Objectbase File

Since the links must be saved in the objectbase file in a two dimensional, serial manner, there must be a method by which the structure of the links in existence can be represented linearly. The implementation of this feature involves the link tags: the link tags of each link from a link attribute is recorded – while the small attribute information is being saved; with each instance and attribute, the owner links are also recorded by tag number.

During the reading of the object file, a record is saved in a hash table as each link structure is read. This table is hashed on the link tag of the link records read, and the size of the hash table is set by the constant "LINK_HASH_TABLE_CONST" in the module r_state.c. Each record in the hash table records the tag of the link read and its owner attribute, i.e. the source of the link.

As each instance and attribute is read, the owner link lists are recreated. All of the fields are set except for the "owner_link_ptr" field, because one cannot guarantee that the source attribute will be read from the objectbase file before the destination object. Finally when all of the instances and attributes have been read from the objectbase file, the links are re-connected.

The reconnection process is quite simple. A pass through every instance and every attribute of the object base is performed. For each object, the owner link lists are traversed, reconnecting the source link for that back link to the object, using add_link. The trick is that the tags in the owner link lists are used to find the record in the hash table corresponding to the link in question. Once this record is found, the owner attribute is known - as well as the destination, and add_link may be called to complete the re-connection procedure.

### 6.11.22 initialize_link_hash_table()

```
static
  int initialize_link_hash_table()


  Description: initializes the link hash table (to NULL.... NULL OK?)
```

### 6.11.23 read_own_link_tags()

```
void
  read_own_link_tags(fp, own_link_tag_list_ptr)
FILE *fp;
OWN_LINK_PTR *own_link_tag_list_ptr;

    This function reads the "back links" recorded in the object file.
    It builds the back links list and places it in the own_link_tag_list_ptr.

Input:
  fp --  file pointer to the open file containing all the intermediate
           form structures which get read.

  own_link_tag_list_ptr -- pointer to the owner links list of some object
         where the back link nodes will be created.
```

```
Output: none
```

### 6.11.24 add_to_hash_list()

```
void
add_to_hash_list(new_hash_node)
     link_hash_node_ptr new_hash_node;
```

This function adds a new node to links hash table. It is called upon reading a link entry from the objectbase file.

### 6.11.25 get_link_attribute_by_tag()

```
ATTRIBUTE_PTR
  get_link_attribute_by_tag(tag)
int tag;
```

This function finds the link_hash entry corresponding to tag, and returns the attribute to which the node points. As a side effect, it deletes the node from the link hash table.

### 6.11.26 read_link_info()

```
void
read_link_info(fp, att)
FILE *fp;
ATTRIBUTE_PTR att;
```

This function is called when a link entry is found in the objectbase file. For each link entry found, it creates a new link hash table entry node, and then adds it to the link table. The link hash table is "hashed" upon the links' tag (id) which is stored with each link in the object base.

```
Input:
  fp --  file pointer to the open file containing all the intermediate
           form structures which get read.

  att --the link's attribute (its owner)

Output: None
```

### 6.11.27 write_own_link_tags()

```
void
write_own_link_tags(fp, own_link_list)
     FILE *fp;
     OWN_LINK_PTR own_link_list;
```

This function is used to write the tags of the back link nodes of each instance and attribute.

### 6.11.28 re_connect_links_list()

```
void
  re_connect_links_list(inst, att)
INSTANCE_PTR inst;
ATTRIBUTE_PTR att;
```

```
  This function reconnects all of the links which point to an instance
  or an attribute.  It is called after all of the instances and objects
  have been read in from the object base file.  The function retrieves
  the link information (i.e. the source attribute) from the link hash
  table, and then adds the link through a call to the standard link
  function: "add_link".  When the link node is retrieved from the
  link_hash_table, it is automatically freed from that table.

  Input: inst:  the instance to which the link should be created -- set
        to NULL if the destination should be an attribute.
 att: the attribute to whgich the link should be linked -- set
        to NULL if the destination should be an instance.
  Output: None
```

### 6.11.29 re_connect_links()

```
void
  re_connect_links(inst)
INSTANCE_PTR inst;

   This function reconnects all of the links which point to a
paritular instance or to any attribute of this instance.  See
re_connect_links_list for the details of this operation.

Input: inst -- the instance in question.

Output: None
```

### 6.11.30 empty_hash_table()

```
int
empty_hash_table()


  This function is just a little test of the hash table routines.
  After the links have been reconnected, the hash table should be
  empty, i.e. every node should be NULL.
  Input: none
Output:


  returns TRUE if the hash table is empty, FALSE otherwise.
```

# Chapter 7

# General Commands

Most of the built in commands in MARVEL can be found in the cmds module. There are specific families of commands that will be covered in later chapters.

## 7.1 Add

These functions all come from the file addinst.c.

### 7.1.1 addinst_cmd()

```
int addinst_cmd(cmd_line)
CMD_LINE_PTR cmd_line;


cmd_line -- the command line arguments

Run the add command.  The name addinst is historical.  If in the graphics
interface, get the arguments, then call do_addinst() to do all the real
work.  When it returns, create all the proper file system type structures,
recalculate the display (for the xface), and go.
```

### 7.1.2 do_addinst()

```
INSTANCE_PTR do_addinst(i_name, a_name)
char *i_name, *a_name;


i_name -- the name of the instance to be added.
a_name -- the name of the attribute to use to hang the instance off of.
          if this is NULL, the inst is added to the current class.
```

```
The meat of the addinst command.  Decide if we are doing a hierarchical
or horizontal addinst, then just do it.  There are numerous cases here,
so just painstakingly take of them.

Notice:  Add to and edit this code with care, it is more complex then it
looks.
```

### 7.1.3 add_hierarchical_instance()

```
INSTANCE_PTR add_hierarchical_instance(i_name, a_name, class)
char *i_name, *a_name;
CLASS_PTR class;


If it has been determined that the instance should be added downward,
this is where the work gets done.  Find the appropriate template
attribute, then the new class.  Then either find or create a attribute
to hang the object off of.  Finally, actually create the object.
```

### 7.1.4 create_and_link_new_instance()

```
INSTANCE_PTR create_and_link_new_instance(i_name, att, class, orig_obj)
char *i_name;
ATTRIBUTE_PTR att;
CLASS_PTR class;
INSTANCE_PTR orig_obj;


i_name -- the name of the instance to be created.
att -- the attribute to hang this instance off of.  If NULL, this is a
       top level instance.
class -- the owner class of the would be instance.
orig_obj -- NULL if this is a new object, otherwise the object which is
            being copied to derive this one.

Create a new object, and link it into the attribute and global lists
of instances.  Create small attributes and link attributes for the new
instance.

If orig_obj is not NULL, then be sure to update the values of all the
small attribtes from it.
```

### 7.1.5 get_unique_inst_name()

```
char *get_unique_inst_name(att, orig_name)
ATTRIBUTE_PTR att;
char *orig_name;
```

att -- the attribute whose list to search;
orig_name -- the name to use as a base;

Generate a unique name based on the given orig_name for an instance that is being copied or renamed. This is done by adding integers, starting with a (static to this function) 0. Thus, with successive executions, you will still get unique names.

### 7.1.6 get_verbose_ok()

```
int get_verbose_ok(i_name, c_name)
char *i_name, *c_name;
```

Check for the verbose flag being on, and ask the user if the addinst should really take place. In the graphics interface, this puts up the yes/no/cancel menu, otherwise it is a query. no and cancel are the same.

i_name and c_name are character strings used to print out a message about actually adding the instance.

### 7.1.7 set_owner_att_stuff()

```
void set_owner_att_stuff(inst, att)
INSTANCE_PTR inst;
ATTRIBUTE_PTR att;
```

Set the owner_att, own_att_tag, and own_att_mclass fields of the instance given. The last two of these are used to recover the objectbase after quitting. The first is for hierarchy traversal. Here inst is hanging off of att.

Notice: upon writing of this comment, I don't quite remember the

```
difference between owner_att_mclass and owner_class. Hmm.
```

The remaining functions for the add command are for handling graphics arguments.

### 7.1.8 addinst_opts_a()

```
int addinst_opts_a(cmd_line, opt)
CMD_LINE_PTR cmd_line;
int opt;

Add the -a option to to the command line, if it was chosen.
```

### 7.1.9 addinst_opts_string()

```
/*ARGSUSED*/
int addinst_opts_string(cmd_line, opt)
CMD_LINE_PTR cmd_line;
int opt;

If the user is doing a hierarchical add, query for an attribute and an
instance, otherwise just an instance name.
```

## 7.2 Change

These functions all come from the file chinst.c.

### 7.2.1 change_cmd()

```
int change_cmd(cmd_line)
CMD_LINE_PTR cmd_line;

cmd_line -- the command line

change the system current object.  Unlike many other commands, this
command operates quite differently in the two interfaces.  Since the
graphics interface has direct access to the instance, the command becomes
trivial in that interface.  Thus the graphics usage is:
```

change <pick>

The display must be valid.

### 7.2.2 change_vert()

```
int change_vert(a_name, i_name)
char *a_name, *i_name;
```

a_name -- the name of an attribute of the current object's class.
i_name -- the name of an object.

These routines sets the system's current object to be
different. Add commands use the notion of current object to operate in.
They are only used in the standard line interface, the graphics interface
uses the display to pick a current object from.

### 7.2.3 change_horiz()

```
int change_horiz(i_name)
char *i_name;
```

Change the current object in the horizontal direction.

### 7.2.4 change_horiz_class()

```
int change_horiz_class(c_name, i_name)
char *c_name, *i_name;
```

Change the current object in the horizontal direction from a class
that is not necessarily the current one. It is found from the given
class name.

### 7.2.5 path_set_prompt()

```
void path_set_prompt(inst, pstr)
INSTANCE_PTR inst;
char *pstr;


inst -- the instance to set the prompt to
pstr -- the optional string.

Set the prompt to either the path to the inst specified, or if null,
to the string given.
```

## 7.3 Load

These functions all come from the file `lum_cmd.c`.

### 7.3.1 load_cmd()

```
int load_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;


Load a strategy.  This is the main calling routine for the loader.
```

### 7.3.2 generate_load_list()

```
int generate_load_list(str_name, keep_file)
     char *str_name, *keep_file;


Used to generate a list of strategies for unloading.

This routine is a bit unusual.  It takes a name
of a strategy to be unloaded, and creates a list of the strategies
remaining in memory which do not have anything to do with that strategy.
Thus, anything which imports the strategy in question will get unloaded
as well.  The user is queried as to whether to go ahead and do the unload
when he or she sees the consequences.  So seperate load and unload lists
are created.  In the end, the load list is checked to see if everything
is available in the current directory, and if so, the list of things
remaining to load is written to disk, and the external loader is called
with that list.
```

### 7.3.3 find_dependent_strs()

```
int find_dependent_strs(head)
     STRLIST_PTR head;
```

Used to generate a list of strategies for unloading.

### 7.3.4 check_readable_and_exists()

```
int check_readable_and_exists(file)
     char *file;
```

file -- name of file to be checked.

Check to see if a file is readable and exists.

### 7.3.5 add_to_strs_list()

```
int add_to_strs_list(head, str)
     STRLIST_PTR head;
     char *str;
```

Used to generate a list of strategies for unloading.

### 7.3.6 free_list()

```
void free_list(list)
     STRLIST_PTR list;
```

Used to generate a list of strategies for unloading.

## 7.4 Unload

These functions all come from the file `lum_cmd.c`.

### 7.4.1 unload_cmd()

```
int unload_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;
```

Unload a strategy. This is the main calling routine to unload strategies with the loader.

### 7.4.2 on_unload_list()

```
int on_unload_list(list, name)
     STRLIST_PTR list;
     char *name;
```

Used to generate a list of strategies for unloading.

## 7.5 Merge

These functions all come from the file lum_cmd.c.

### 7.5.1 merge_cmd()

```
int merge_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;
```

Merge a strategy. This is the main calling routine for the loader in merging mode.

## 7.6 Save

These functions all come from the file save_cmd.c.

### 7.6.1 save_cmd()

```
int save_cmd(cmd_line)
```

```
     CMD_LINE_PTR cmd_line;


cmd_line -- structure containing user command and its arguments.

This routine saves the current status of the objectbase and the
environment (i.e., the loaded strategies and other environment
variables.  The user has a choice of only saving the status of the
objectbase, only the environment or both.  The routine calls
write_objectbase() and write_strategies() to save the appropriate
status.
```

The next two functions just handle graphics arguments to the save command.

### 7.6.2 save_opts_both()

```
/*ARGSUSED*/
int save_opts_both(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;


cmd_line -- structure containing user command and its arguments.
```

### 7.6.3 save_opts_single()

```
int save_opts_single(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;


cmd_line -- structure containing user command and its arguments.
```

## 7.7 Set

These functions all come from the file `set_cmd.c`.

### 7.7.1 set_cmd()

```
int set_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;
```

```
This is the central displaying and setting routine.  It is rather long
but just does the same thing for each possible thing to set.  Note that
it acts one way for things that are boolean, and only expects one
corresponding argument, and another for things that take values, where
there are two corresponding values.
```

The remainder of the functions dealing with the set command just handle the recieving of graphics arguments.

### 7.7.2 get_set_graphic_args()

```
int get_set_graphic_args(cmd_line)
     CMD_LINE_PTR cmd_line;


   something
```

### 7.7.3 set_opts_choose_font()

```
int set_opts_choose_font(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

### 7.7.4 set_opts_depth()

```
int set_opts_depth(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;


   something
```

### 7.7.5 set_opts_no_choose()

```
/*ARGSUSED*/
int set_opts_no_choose(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;


   something
```

### 7.7.6 set_opts_show_all()

```
/*ARGSUSED*/
int set_opts_show_all(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;


   something
```

## 7.8 Quit

These functions all come from the file `commands.c`.

### 7.8.1 quit_cmd()

```
int quit_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;

    cmd_line -- structure containing user command and its arguments.
```

The remainder of the functions dealing with the **quit** command just handle the recieving of graphics arguments.

### 7.8.2 quit_opts_cancel()

```
/*ARGSUSED*/
int quit_opts_cancel(cmd_line, opt)
```

```
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

### 7.8.3 quit_opts_n()

```
int quit_opts_n(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

### 7.8.4 quit_opts_s()

```
/*ARGSUSED*/
int quit_opts_s(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

## 7.9 Readob

These functions all come from the file commands.c.

### 7.9.1 readob_cmd()

```
/*ARGSUSED*/
int readob_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;

    cmd_line -- structure containing user command and its arguments.
```

## 7.10 Help

These functions all come from the file commands.c.

### 7.10.1 help_cmd()

```
int help_cmd(cmd_line)
     CMD_LINE_PTR cmd_line;

    cmd_line -- structure containing user command and its arguments.
```

The remainder of the functions dealing with the help command just handle the recieving of graphics arguments.

### 7.10.2 help_opts_command()

```
int help_opts_command(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

### 7.10.3 help_opts_quest()

```
int help_opts_quest(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

### 7.10.4 help_opts_subject()

```
/*ARGSUSED*/
int help_opts_subject(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

    cmd_line -- structure containing user command and its arguments.
```

## 7.11 Links

These functions come from the file `link.c`.

### 7.11.1 link_cmd()

```
int
link_cmd(cmdLine)
     CMD_LINE_PTR cmdLine;
```

cmdLine -- structure containing user command and its arguments.

description: the user-level command for initiating a link from a source (an attribute of link_type) to a destination (an attribute or an instance).

### 7.11.2 unlink_cmd()

```
int
unlink_cmd(cmdLine)
     CMD_LINE_PTR cmdLine;
```

input: cmdLine -- structure containing user command and its arguments.

description: this command unlinks (aka "removes", "deletes") a link given the source and destination objects (for the command line interface), or after polling the user for the source and dest. objects.

output: TRUE/FALSE, depending upon whether the unlinking was sucessfull.

# Chapter 8

# Organizational commands

The organ module contains commands to organize Marvel objectbases. The best backgound documentation for these commands is found in [Sok89].

## 8.1 Background Functionality

### 8.1.1 check_reset_cur_instance()

```
void check_reset_cur_instance(inst)
INSTANCE_PTR inst;

Check to be certain that we have the current instance set to something
other then the inst we are removing.  Reset as follows:

    prev, next, owner_att->owner_instance, first inst of dbroot, NULL.

inst is the instance which is currntly in question (not necessarily the
current one.
```

### 8.1.2 copy_inst_disk_structures()

```
void copy_inst_disk_structures(from, to)
char *from, *to;

Use this routines to copy a directory to another.
The later will be placed under the former.

from -- the disk object to be copied.
```

```
to -- the target disk object.
```

### 8.1.3 move_inst_disk_structures()

```
void move_inst_disk_structures(from, to)
char *from, *to;


Use this routines to move a directory to another.
The later will be placed under the former.

from -- the disk object to be copied.
to -- the target disk object.
```

### 8.1.4 remove_inst_disk_structures()

```
void remove_inst_disk_structures(path)
char *path;


Call this routine after an instance in the objectbase is completely
deleted.  Currently, just do a system rm -rf of the directory in
question, this will doubtless change soon.

path is the path to delete.  It is found via a concatenation of names
in the db.
```

### 8.1.5 remove_large_attribute()

```
void remove_large_attribute(att)
ATTRIBUTE_PTR att;


Remove attribute removes a large attribute of an instance when there is
no longer any need for it.  This means that there are no more instances
hanging off that large attribute.
```

## 8.2 Copy

### 8.2.1 copy_atts()

```
void copy_atts(from_inst, to_inst)
INSTANCE_PTR from_inst, to_inst;
```

from_inst -- the source
to_inst -- the one getting the new atts

Using the source instance as a template, copy over all the attributes of that instance to the target one.

### 8.2.2 copy_cmd()

```
int copy_cmd(cmd_line)
CMD_LINE_PTR cmd_line;
```

This is the main entry point for the copy command. Unlike many built in commands, copy (and the rest of the organ commands, do not build up a command line in the graphics interface, rather they just get the info needed and go. The line interface is handled seperately.

The reason for this is that the graphics versions can loop, doing many copies until done is pressed. Thus it would be awkard to be building this unnatural command string within each part of the loop.

In the graphics interface, this command works as follows:

copy [<from_pick> <to_pick>] ... <done_pick>.

### 8.2.3 copy_tree()

```
void copy_tree(from_inst, to_inst)
INSTANCE_PTR from_inst, to_inst;
```

from_inst -- the source
to_inst -- the matching new guy.

Copy all instances and attributes of from_inst to to_inst. These are instances at the same hierarchical level. Do this by first copying all attributes, then getting each child instance, recursively.

### 8.2.4 do_copy()

```
INSTANCE_PTR do_copy(from_inst, to_inst)
INSTANCE_PTR from_inst, to_inst;
```

```
from_inst -- the instance to start the copy from.  It will get copied,
             as well as any children.
   to_inst -- the instance we will copy to.
```

This is the heart of the instance copying routine.

As a first pass, there will be the following restrictions for a successful copy.

1. from_inst must be logically on class below to_inst.
2. to_inst can not be from_inst's parent.

This routine returns the new instance that it creates. This should be used to get the possibly new name for doing the file system stuff. NULL is returned if the routine fails.

## 8.3 Join

### 8.3.1 do_join()

```
int do_join(from_inst, to_inst)
INSTANCE_PTR from_inst, to_inst;
```

```
   from_inst is the instance to start the join from.
   to_inst is the instance we will join to.

   This is the heart of the instance joining routine.  This routine has a
   possible unpleasent side effect of leaving some instances moved and some
   not in the event of a failure in the middle of the command.  Such a
   failure "should not" happen, of course.
```

As a first pass, there will be the following restriction for a successful join.
1. from_inst must be logically from same class as to_inst.

### 8.3.2 join_cmd()

```
int join_cmd(cmd_line)
CMD_LINE_PTR cmd_line;
```

This is the main entry point for the join command.

The idea here is to join all the elements of one object to those of another class. In concept, the classes can be different, for a first pass, however, we will impose a same class restriction upon them.

In the graphics interface, this command works as follows:

join [<source> <target> ... ] <done_pick>.

## 8.4 Delete

### 8.4.1 delete_cmd()

```
int delete_cmd(cmd_line)
CMD_LINE_PTR cmd_line;
```

This is the main entry point for the delete command.

In the graphics interface, this command works as follows:

delete [<pick> ... ] <done_pick>.

### 8.4.2 do_delete()

```
int do_delete(inst)
INSTANCE_PTR inst;
```

This is the heart of the instance deletion routine.

```
inst -- the instance to start the delete from.  It will get
        deleted, as well as any children.

We must delete from the bottom up, do a recursive traversal of the
tree below inst, and delete on the way "back up" the tree.
```

When deleting an instance, the links which point to that instance (or to any of its attributes) and the links which originate from any of the attributes of the instance must also be removed. This process creates some interesting problems with terminology as one must write about deleting a link (while unlinking is often used to described the process of removing an object from a link list). The function below which unlinks an instance (i.e. deletes an instance) calls the link module function: "remove_link_to_from_inst" which is described in the section on Graphical Links.

### 8.4.3 unlink_inst()

```
int unlink_inst(inst)
INSTANCE_PTR inst;

   inst -- the instance to be unlinked.  It MUST be a leaf level
           instance, or this routine will fail.

   unlink the instance in question.  Fix up prev and next, and global prev and
   global next.  Be careful about heads of lists, and instances on the top
   level.

   1/23/90: also, remove all links which point to this instance, or to
   this instance's attributes.
```

## 8.5 Move

### 8.5.1 do_move()

```
INSTANCE_PTR do_move(from_inst, to_inst)
INSTANCE_PTR from_inst, to_inst;


from_inst is the instance to start the move from.  It will get moved,
          as well as any children.
to_inst is the instance we will move to.
```

This is the heart of the instance moveing routine. For the objectbase, a move is accomplished by copying the stuff in question, then deleting the old stuff. It is doubtless more efficient to implement move as a relinking of the top level pointer, I might do this soon.

As do_copy(), this routine returns the new instance created. Note the name only might be the same as the one copied, due to to the auto resolving of names.

As a first pass, there will be the following restrictions for a successful move.

1. from_inst must be logically on class below to_inst.
2. to_inst can not be from_inst's parent.

### 8.5.2 move_cmd()

```
/*ARGSUSED*/
int move_cmd(cmd_line)
CMD_LINE_PTR cmd_line;
```

This is the main entry point for the move command.

Move is identical to copy, with the exception that things are not duplicated, but rather moved.

In the graphics interface, this command works as follows:

move [<source> <destination> ... ] <done_pick>.

## 8.6 Rename

### 8.6.1 do_rename()

```
int do_rename(cmd_line, inst)
CMD_LINE_PTR cmd_line;
INSTANCE_PTR inst;
```

The meat of the rename command. Remember to free the old name. We can

just do a redraw of the display here, without any recalculations.

### 8.6.2 do_rename_disk_structures()

```
void do_rename_disk_structures(old, new)
char *old, *new;
```

old -- the old name
new -- the new name

Rename the actual disk structures associated with the instance in question. Note that renaming is effectively a unix mv.

### 8.6.3 rename_cmd()

```
/*ARGSUSED*/
int rename_cmd(cmd_line)
CMD_LINE_PTR cmd_line;
```

This is the main entry point for the rename command.
In the graphics interface, this command works as follows:

rename [<pick> <string> ... ] <done_pick>.

# Chapter 9

# Evaluation of Rules

The evaluator module contains code to evaluate arbitrarily complex preconditions and and assert arbitrarily complex postconditions in the MARVEL process model. All the involved code is contained within the evaluator module.

## 9.1 Evaluating Preconditions

The functions directly involved in evaluation preconditions in the objectbase, to determine if an activity is ready for execution, are all found in the file eval_pre.c. These functions are all read only on the objectbase, however they create their own internal datastructures.

### 9.1.1 eval_pre()

```
int eval_pre(rule)
RULE_PTR rule;


Evaluate the preconditions of the rule.

This routine first builds a set of objects to apply the property list
to.  It then applys that property list, thus determining if chaining
needs to be done.  The status is printed for successful or chaining
required type of evaluations.

eval_pre(), and all the gems it calls, are read only with respect to
all but it's own data structures, and the two failed fields of the rule
described below.  All it's datastructures are basically recreated for
each evaluation, because there is no way of knowing what postconditions
might have been asserted (or objects added, for that matter, since
the last evaluation of the rule in question, on the parameters in
question.  We have discussed this in detail, but have not come up
```

with satisfactory ways to save computations from one evaluation to the next without unduly complicating overly complex stuff.

SATISFIED is returned if the precondition is satisfied. If there was some sort of internal or data structure related error, INTERNAL ERROR is returned. If any of the bindings were empty sets, EMPTY_BINDINGS is returned. This is not necessarily desireable, but it makes it so a rule can not do something such as try to compile a module that has no preconditions. If chaining needs to be done, DO_CHAIN is returned, and the failed predicate is placed in the rule's failed_pred field. The object for which it failed (important!) is placed in the rule's failed_obj field.

Disclaimer: The code that this routine goes throught is rather complex. Before modifying it, the reader should have a fairly good command of what in the world all this stuff is doing. The comments in the routines are reasonable, but the datastructures need to be throughally understood to make any sense of all this. On the bright side, this is the only place that this stuff is called, and it is only called from a few places in the opportunist.

### 9.1.2 build_characterized_binding_list()

```
int build_characterized_binding_list(rule)
RULE_PTR rule;
```

rule -- the rule in question

Walk through a rule's bindings, and build a complete list of objects that are bound to this execution of the rule. This is done by calling the much more complex routine build_indiv_binding() for each of the bindings in the rule.

### 9.1.3 build_indiv_binding()

```
int build_indiv_binding(rule, binding)
RULE_PTR rule;
BINDING_PTR binding;
```

This routine takes an individual binding clause, and builds a list of bindings for it, which it leaves on the appropriate symbol in the list of

symbols for that rule. These symbols might already have been created, if so, this is used to define scope in the binding. If there is an unreferenced symbol, it is an error, and INTERNAL_ERROR is returned. Otherwise, SATISFIED is returned.

Rule is used to avoid having to save back pointers from the binding.

### 9.1.4 check_obj_against_cfunc()

```
static int check_obj_against_cfunc(rule, obj, binding, symb, cond)
RULE_PTR rule;
INSTANCE_PTR obj;
BINDING_PTR binding;
SYMBOL_PTR symb;
COND_PTR cond;
```

```
rule -- the rule in question
obj -- the object in question
binding -- the binding that contains the characteristic function.
symb -- the symbol that is used in the binding.
cond -- the cond where we are in the recursion.
```

This routine checks to see if the given object meets the characterizing function assigned to it. This function will most often be a member type of function.

It works recursively for complex characteristic functions (with and, or and not). When recursing, cond is the pointer to the cond to be dealt with, in the first call to the routine, cond should be NULL.

TRUE is returned is the object is valid, and FALSE otherwise. FALSE means the membership test failed, and INTERNAL_ERROR means just that.

### 9.1.5 eval_prop_list()

```
int eval_prop_list(rule, cond)
RULE_PTR rule;
COND_PTR cond;
```

```
rule -- the rule whose property list needs checking.
cond -- the location in the traversal.
```

This routine recursively descends the property list, checking each property for truth. It checks the appropriate logical relationships between the properties, and returns TRUE or FALSE depending upon the outcome..

The rules's failed_pred_index is set somewhere in the depths of this procedure after the first failure. NOT is not yet implemented.

### 9.1.6 check_property_value()

```
int check_property_value(rule, cond)
RULE_PTR rule;
COND_PTR cond;
```

This routine starts the heart of the evaluation process. Cond is a single condition that needs to be verified against all the possible objects in the rule's symbol table's runtime_objs. So loop through all these runtime objects.

The first side of the property will always be a BVAR (?<varname>.<attname>). So extract the variable name and the attribute name from the predicate, The other half of what to compare against is found in do_comparison(). After that routine is executed, check to see if the first symbol is a PARAMETER or BINDING. If it is a PARAMETER this will be the only object in the list, so return SATISFIED if the comparison was successfull, otherwise DO_CHAIN. If this was a BINDING, check the quantifier to see if we need to continue comparisons. If so, do all the comparisons, keeping the result. Otherwise return DO_CHAIN or SATISFIED, as appropriate. Finally, recheck the kinding of quantifier we are working with, and return what is proper. Think hard about why we can stop comparisons early some of the time!

### 9.1.7 do_comparison()

```
int do_comparison(rule, pred, obj1, symbol1, a_name1)
RULE_PTR rule;
PRED_TABLE_PTR pred;
INSTANCE_PTR obj1;
SYMBOL_PTR symbol1;
char *a_name1;
```

Compare two sides of a predicate. The important thing here is to recognize all the possible cases of comparison. The right hand side can either be a simple int, string, time stamp, double or username, or it can be another bvar. So check what it is, and do the correct low level comparison.

INTERNAL_ERROR is returned if anything is funky with the data structures. Otherwise the value of the comparison (TRUE or FALSE) is returned.

### 9.1.8 eval_print_status()

```
void eval_print_status(rule, result)
RULE_PTR rule;
int result;
```

rule -- the rule we are dealing with
result -- TRUE is we can fire the rule, and FALSE otherwise

This rule prints a nice message that says whether we can execute this rule. This message becomes part of the text window's dialogue that represents what the system is doing when starting to chain.

### 9.1.9 get_all_bound_objects()

```
int get_all_bound_objects(class, rule, binding, symb)
CLASS_PTR class;
RULE_PTR rule;
BINDING_PTR binding;
SYMBOL_PTR symb;
```

For all the objects in the given class, look at each one and see if it satisfies the condition in the characteristic function at hand. If so, add it to the given symbol's list of runtime objects, which is used to verify the property list part of the precondition.

Once this is done for all the instances of this class, do it for all the specializations of this class (the subclasses of the symbol's class. Note that class is only the same as symb->class for the first level of recursion, thereafter class becomes the specialization to

```
check for.

Return INTERNAL_ERROR for any small little thing which is out of order,
otherwise return SATISFIED.  Note that a SATISFIED does not especially
imply that any objects have been bound, but rather that everything
worked out alright.
```

## 9.2 Asserting Postconditions

The functions directly involved in asserting postconditions in the objectbase, as the result of some completed activity, are all found in the file `eval_post.c`

### 9.2.1 assert_posts()

```
COND_PTR assert_posts(rule, which_post)
RULE_PTR rule;
int which_post;

   rule -- the rule that has postconditions to be asserted
   which_post -- which post to assert

   This routine finds the proper post condition to be asserted.  It then
   asserts each property it finds for each object appropriate in the
   rule's list of symbols.  The post condition that was asserted is
   returned, or NULL if there was a failure.
```

### 9.2.2 assert_property()

```
static int assert_property(rule, cond)
RULE_PTR rule;
COND_PTR cond;


This routine is called for each property that assert_posts() has found
to assert.  It could therefore be called several times for a single
postcondition that is an and of several properties.  Thus cond is
one simple property.

It is assumed that the right hand side will be a BVAR (?var.attribute),
and furthermore a parameter to the rule, rather then some bound variable.
This is part of the semantic definition of asserting properties.  The
```

left hand side is examined, it may either be a value or another BVAR (not necessarily a parameter). The appropriate value is extracted from the left hand side, and asserted into the right hand side. The change flag of the cond is set to signify if there really was a change in the value, this affects the opportunists future chaining.

### 9.2.3 assert_att_int()

```
int assert_att_int(cond, att, value)
COND_PTR cond;
ATTRIBUTE_PTR att;
int value;
```

Assert an integer type value into the given attribute. This routine is called in the process of asserting postconditions.

### 9.2.4 assert_att_real()

```
int assert_att_real(cond, att, value)
COND_PTR cond;
ATTRIBUTE_PTR att;
double value;
```

Assert a double type value into the given attribute. This routine is called in the process of asserting postconditions.

### 9.2.5 assert_att_string()

```
int assert_att_string(cond, att, value)
COND_PTR cond;
ATTRIBUTE_PTR att;
char *value;
```

Assert a string type value into the given attribute. This routine is called in the process of asserting postconditions.

### 9.2.6 assert_att_values()

```
int assert_att_values(cond, att1, att2)
COND_PTR cond;
ATTRIBUTE_PTR att1, att2;

Assert the value in the second attribute into the first attribute.
First make certain that the types are compatible.
```

## 9.3 Extracting and Comparing Values

Extraction functions are used to extract parts of attributes and variables. They are all found in the file `extract.c`.

### 9.3.1 extract_aname()

```
char *extract_aname(b_var)
char *b_var;

   b_var -- the variable.

   the format of a b_var is ?<var_name>.<att_name>.  This routine returns
   a malloced string corresponding to the att_name.
```

### 9.3.2 extract_var()

```
char *extract_var(b_var)
char *b_var;

   *b_var -- the variable.

   the format of a b_var is ?<var_name>.<att_name>.  This routine returns
   a malloced string corresponding to the var_name, including the ?.
```

### 9.3.3 is_this_a_bvar()

```
int is_this_a_bvar(b_var)
char *b_var;

   b_var -- the variable.

   the format of a b_var is ?<var_name>.<att_name>.  This routine returns
   a malloced string corresponding to the var_name, including the ?.
```

Comparison functions are used to compare values of attributes to each other to determine the value for sub–evaluations. There is a comparison function for each of the major types of small attributes. They are all found in the file compare.c.

### 9.3.4 compare_att_values()

```
int compare_att_values(att1, att2, operator)
ATTRIBUTE_PTR att1, att2;
int operator;


Determine the results of a comparison of the type:

  (?a.att  op  ?b.att)  where op is an appropriate operator, and
                        ?a and ?b are variables previously defined in
                        the rule.

The attributes must be of the same type.
```

### 9.3.5 compare_doubles()

```
int compare_doubles(double1, double2, operator)
double double1, double2;
int operator;


Compare two doubles, based upon the kind of operator given.  If the
comparison comparison succeeds, return TRUE.  Otherwise, either the
comparison failed or the operator was incompatible.  In the later case,
a message is printed.  FALSE is returned in both cases.

Acceptable operators are NEQ, EQ, GEQ, LEQ, GT and LT.
```

### 9.3.6 compare_ints()

```
int compare_ints(int1, int2, operator)
int int1, int2, operator;
```

Compare two integers, based upon the kind of operator given. If the comparison comparison succeeds, return TRUE. Otherwise, either the comparison failed or the operator was incompatible. In the later case, a message is printed. FALSE is returned in both cases.

Acceptable operators are NEQ, EQ, GEQ, LEQ, GT and LT.

### 9.3.7 compare_strings()

```
int compare_strings(str1, str2, operator)
char *str1, *str2;
int operator;
```

Compare two strings, based upon the kind of operator given. If the comparison comparison succeeds, return TRUE. Otherwise, either the comparison failed or the operator was incompatible. In the later case, a message is printed. FALSE is returned in both cases.

Acceptable operators are NEQ and EQ. We do not currently support LT and GT for strings. It could be easily added here is someone saw reason.

### 9.3.8 compare_times()

```
int compare_times(time1, time2, operator)
double time1, time2;
int operator;
```

Compare two times, based upon the kind of operator given. If the comparison comparison succeeds, return TRUE. Otherwise, either the comparison failed or the operator was incompatible. In the later case, a message is printed. FALSE is returned in both cases.

If either time is the special time -1, then use the current time for it.

Since that is in microseconds, be careful about getting both at the same time.

Acceptable operators are NEQ, EQ, GEQ, LEQ, GT and LT.

### 9.3.9 compare_users()

```
int compare_users(u1, u2, operator)
char *u1, *u2;
int operator;
```

Compare two users, based upon the kind of operator given. If the comparison comparison succeeds, return TRUE. Otherwise, either the comparison failed or the operator was incompatible. In the later case, a message is printed. FALSE is returned in both cases.

If u2 is the string "CurrentUser", then u1 will be compared with who ever is the current user.

Acceptable operators are NEQ and EQ. It could be easily added here is someone saw reason.

# Chapter 10

# Opportunistic Processing

The opportunist module contains code to implement opportunistic processing amongst the rules of a Marvel process model.

## 10.1 Calling the Executer

The following routines are from the file call_sched.c

### 10.1.1 call_scheduler()

```
int call_scheduler(rule)
RULE_PTR rule;


Call the scheduler with the arguments in the rule's activity's
arguments runtime objects.  This is all turned into an argv
type array here.
```

## 10.2 Backward Chaining

The following routines are from the file chain_back.c

### 10.2.1 do_backward_chain()

```
int do_backward_chain(rule)
     RULE_PTR rule;


rule -- the rule whose precondition is not satisfied.  The
```

failed_pred field of the rule contains the actual
predicate that is not satsfied.

This function tries to satisfy a precondition by backward chaining. The routine calls get_all_rules_in_back_chain() to get all the rules that can potentially satisfy the failed predicate. It then calls put_rules_in_back_and_exec_queues() to divide these rules into those whose preconditions are satisfied (and thus they are ready to be executed) and those who would need further backward chaining. Two queues are created that will try to accomplish that task. The first queue is the execution queue. This queue will contain the rules that can satisfy the failed predicate and whose preconditions are true. The other queue is the backward queue. This queue contains rules that can satisfy the failed predicate but whose preconditions are false themselves.

If a rule on the execution queue can satisfy the precondition then the backward queue will be ignored. On the other hand, if all the rules on the execution queue could not satisfy the failed precondition, the backward queue will be used to call this routine recursively to try and satisfy their preconditions first and then the original failed precondition. If the precondition is satisifed, TRUE is returned, otherwise FALSE

### 10.2.2 exec_rules_on_back_que()

```
int exec_rules_on_back_que(que, rule)
     QUEUE_PTR que;
     RULE_PTR rule;
```

que -- pointer to the backward queue

rule -- the original rule whose precondition we are trying to satisfy.

All the rules on the backward queue are not satisfied, so this routine calls do_backward_chain() on each one of them in order. If the backward chaining succeeded in making the precondition of the rule on the backward queue satisfied, the routine executes the rule. It then checks if the original failed predicate is now satisfied. If it is, TRUE is returned. If it didn't, it tries the next rule on the backward queue. If none of the rules winds up satisfying the predicate, FALSE is returned to indicate that the predicate cannot be satisfied.

### 10.2.3 exec_rules_on_exec_que()

```
int exec_rules_on_exec_que(que, failed_rule)
     QUEUE_PTR que;
     RULE_PTR failed_rule;


que -- pointer to the execution queue.

failed_rule -- the rule that initiated the current backward cahining
               cycle.

This routine executes rules from the execution queue passed to it.  After
each rule has been executed, it checks if the original failed predicate
is now satisfied.  If it is, TRUE is returned.  If none of the rules
winds up satisfying the predicate, FALSE is returned.
```

### 10.2.4 get_all_rules_in_back_chains()

```
INFO_LEVEL_PTR get_all_rules_in_back_chains(rule)
     RULE_PTR rule;


rule -- This is the rule that has a predicate that is not
        satisfied.

This function will create a linked list of rules that can possibly
satisfy the failed predicate that is passed to the routine
```

### 10.2.5 put_rules_in_back_and_exec_queues()

```
void put_rules_in_back_and_exec_queues(level, execution_que, backward_que,
failed_rule)
     INFO_LEVEL_PTR level;
     QUEUE_PTR execution_que,backward_que;
     RULE_PTR failed_rule;


level -- This structure contains the rules and information
         necessary to satisfy a precondition at a
         particular level in the backward chaining tree.
```

```
execution_que --  This structure will hold the rules that
                  can be executed that this moment which can be satisfy
                  precondition in question.

backward_que -- This structure will hold the rules whose
                preconditions are false. The preconditions for these
                rules must be satisfied through backward chaining.

failed_rule -- the rules that caused the backward chaining.

This routine goes through the level linked list and puts all the rules
whose preconditions are satisfied on the execution queue and those
whose preconditions are not satisfied on the backward queue.
```

### 10.2.6 satisfy_pre()

```
int satisfy_pre(post, predicate)
     COND_PTR post;
     PRED_TABLE_PTR predicate;


posts --  The postcondition which was asserted by the evaluator.

predicate -- the predicate that is currently not satisfied

This routine checks if the postcondition passed to it satisfies the
predicate that is also passed to it.  The idea is that after a rule
is executed in a backward chaining cycle, we want to check if the rule
actually did satisfy the original failed predicate.
```

## 10.3 Forward Chaining

The following routines are from the file chain_forward.c

### 10.3.1 do_forward_chain()

```
int do_forward_chain(postconds)
   CHAINS_PTR postconds;

   postconds -- This structure contains all the information
                neccessary to perform forward chaining on a
postcondition
```

This routine does forward chaining if the postconditions of a rule chain to other rules. A linked list of structures containing all the posconditions of a rule will be sent down to this routine. Each structure contains a a postcondition (a list of predicates) and a pointer to the initial rule that started the forward chaining. The rules that can be executed will then be executed and their postconditions will be asserted. This routine then creates it's own list of structures as each rule's postcondition is asserted so that it can make a recursive call to process the next postconditions.

### 10.3.2 get_all_rules_in_forward_chains()

```
INFO_LEVEL_PTR get_all_rules_in_forward_chains(postconds)
     CHAINS_PTR postconds;

   postconds -- The asserted postconditions of the rules that have
                successfully executed in the last chaining cycle.
```

This routine creates a linked list of structures that contain pointers to rules that will be used in forward chaining. It goes through each postcondition in the list passed to it. For each postcondition, it goes through each predicate that was asserted and extracts the rules that the predicate causes forward chaining to. If the rules are not on the list already, it inserts them.

### 10.3.3 put_rules_in_execution_queue()

```
void put_rules_in_execution_queue(level,execution_que)
    INFO_LEVEL_PTR level;
    QUEUE_PTR execution_que;

   level -- This is a linked list of structures that contains
            all the rules that are on the forward chaining
    list of a postcondition that has been asserted

   execution_que --  This queue will hold all the rules whose
                     preconditions are true.
```

This routine creates an execution queue of rules that are ready to be executed in a forward chaining cycle. It takes a structure that is

produced by get_all_rules_in_forward_chains (containing all the potential forward chaining rules). From all of the potential rule, only those whose precondition is true will be added to the execution queue. No duplicate rules will be added to the queue. A list of executed rules will be kept

## 10.4 Handling the Arguments to Rules

The following routines are from the file handle_args.c

### 10.4.1 compare_runtime_objs()

```
int compare_runtime_objs(symbols1, symbols2)
     SYMBOL_PTR symbols1, symbols2;
```

```
symbols1, symbols2 -- the two symbol tables, whose runtime objects we
                      will compare
```

This routine checks if the runtime objects of the symbol table entries of type PARAMETER are the same in the two symbol tables. This is used to check if two instances of the same rule are identical (i.e., their activities will do the exact same thing on the exact same objects).

### 10.4.2 get_obj_from_symbols()

```
INSTANCE_PTR get_obj_from_symbols(class, symbols)
CLASS_PTR class;
SYMBOL_PTR symbols;
```

```
class - the class of the object that we want to try to extract from
        symbols

symbols - the symbol table from which we want to find an object whose
          class is either the same or a subclass of the class argument
```

This routine tries to find a runtime object from the symbol table passed to it, where the class of the object is a subclass of the class passed to the routine. Unlike lookup_arg() above, it doesn't return an entry in the symbol table, but a pointer to an actual object in the objectbase. It uses the symbol table to find out how to search the

composite objet hierarchy starting with the curren object.

### 10.4.3 handle_args()

```
int handle_args(rule, cmd_line)
     RULE_PTR rule;
     CMD_LINE_PTR cmd_line;
```

rule -- The rule whose symbol table will hold the runtime
objects that the entries of the cmd_line param. point to.

cmd_line -- a list of structures containing the runtime objects that
the user entered as arguments to the command corresponding
to the rule passed.

This routine places the runtime objects passed in cmd_line into the correct entries of the symbol table of the rule. An error is returned if the number of arguments in cmd_line is different from the number of entries whose type is PARAMETER in the symbol table. An error is also returned if the types of the runtime objects passed and the types of the symbol table entries do not match.

### 10.4.4 lookup_arg()

```
SYMBOL_PTR lookup_arg(class, symbols, var)
     CLASS_PTR class;
     SYMBOL_PTR symbols;
     char *var;
```

symbols -- The symbol table that the routine will search.

class -- The class of the variable we want to extract from symbol table.

var -- the name of the variable we want to extract from symbol table.

This routine tries to extract a variable from symbols whose name is the same as var, and whose class is either the same as the class passed to it or a subclass of it. If none if found, NULL is returned.

### 10.4.5 rule_has_args()

```
int rule_has_args(rule)
     RULE_PTR rule;
```

rule -- the rule whose parameters we need to check

This routine checks if a rule has arguments. If it does, it returns TRUE, otherwise, it returns FALSE. Note that this replaces the rule->proc_flag in the old code. Also note that this does not say anything about the different activities that a rule might have, the loader potentially screwed this up before.

### 10.4.6 set_arg_rule()

```
int set_arg_rule(rule, symbols, predicate, chaining)
     RULE_PTR rule;
     SYMBOL_PTR symbols;
     PRED_TABLE_PTR predicate;
     int chaining;
```

rule -- The rule that was either just executed or whose precondition failed and thus caused chaining.

symbols - The symbol table of the rule that we will chaingoing to be used for backward or forward chaining.

predicate - The predicate that caused the forward chain between rule and the forward rule whose symbol table is passed.

chaining - flag to indiciate whether this routine is called during forward or backward chaining.

This routine will pass the arguments from one rule to another rule using their symbol tables. The arguments are passed from one rule to another only if their class definitions in the symbol table entries for each rule are the same and if the entries are arguments to the rule.

## 10.5 Managing the Opportunist Lists

The following routines are from the file `list_manage.c`

### 10.5.1 add_rule_to_list()

```
void add_rule_to_list(rule, back_or_exec, failed_rule)
     RULE_PTR rule;
     int back_or_exec;
     RULE_PTR failed_rule;

   rule -- pointer to the rule that  will be added to
           the list specified by the back_or_exec flag.

   int back_or_exec - which list the rule name should be added to.

   This routine adds a rule to one of the three list:
      1) Back_list - all the rules that are possibly going to be used for
                     backward chaining.

      2) execution_list - all the rules that have been successfully executed.

      3) Rule_list - a list of all the rules encountered in forward chaining.
```

### 10.5.2 free_entire_list()

```
void free_entire_list(list)
     int list;

   list -- which one of the three global lists to free?

   This routine frees up either the execution, forward, or back list.
```

### 10.5.3 init_list()

```
void init_list(which_list)
     int which_list;

   which_list - which one of the three global lists: execution_list,
```

```
                rule_list or back_list;

  This routine initializes one of the three global list.
```

### 10.5.4 look_uplist()

```
int look_uplist(rule,back_or_exec, failed_rule)
   RULE_PTR rule;
   int back_or_exec;
   RULE_PTR failed_rule;

   rule -- This rule we will search for in one of the lists.

   back_or_exec --  which list to lookup rule in

   failed_rule -- the rule that caused the chaining cycle to start

   This routine checks one of three list for a rule_name:
    1) execution_list
    2) back_list
    3) rule_list

    TRUE is returned if the rule is found, FALSE otherwise
```

## 10.6 Predicate Comparisons

The following routines are from the file match_pred.c

### 10.6.1 do_operands_match()

```
int do_operands_match(op1, op2, rule1, rule2)
     char * op1, *op2;
     RULE_PTR rule1, rule2;

   op1, op2 -- the two operands that need to be checked.

   rule1 -- the rule to which op1 belongs

   rule2 -- the rule to which op2 belongs

   This routine checks if op1 = op2.  It checks whether the two operands
```

```
   belong to the same class and whether their attributes are the same.
   If both are true, it returns TRUE; otherwise, returns FALSE
```

### 10.6.2 do_preds_match()

```
int do_preds_match(pred1, pred2)
     PRED_TABLE_PTR pred1, pred2;

   pred1 - a predicate in a postcondition of a rule.

   pred2 - a predicate in a precondition of a rule.

   This routine checks whether pred1 can chain to pred2.

   if the predicate is of the form:
              BVAR1 = BVAR2   or  BVAR1 = string

   any predicate of the form

               BVAR1 = BVAR3 or  BVAR3 = BVAR1 or BVAR1 = string or
               BVAR1 <> FOO   where FOO is <> BVAR2 or string

   should cause chaining
```

## 10.7 The Opportunist

The following routines are from the file opportun.c

### 10.7.1 cleanup_rule()

```
void cleanup_rule(rule)
     RULE_PTR rule;


rule - a pointer to the rule whose symbol table must be cleaned up.

This routine goes through the symbol table of rule and for each entry
of type PARAMTER, it clears the runtime objects of the parameter.  It
also removes from the symbol table all the entries that are not
parameters (i.e., bindings that were inserted while evaluating the
rule.
```

The routine also resets the failed_pre_index and failed_object fields of the rule.

### 10.7.2 create_rule_instance()

```
RULE_PTR create_rule_instance(rule)
     RULE_PTR rule;
```

rule - the original rule which we want to duplicate.

This routine creates a copy of the rule passed to it and returns a pointer to this copy. This is needed in order to execute multiple instances of the same rule in the same chaining cycle. Each instance will have its own symbol table and its own runtime objects.

### 10.7.3 process_rule()

```
int process_rule(rule, cmd_line)
   RULE_PTR rule;
   CMD_LINE_PTR cmd_line;
```

rule -- The rule that is going to be executed.

cmd_line -- a list of structures containing the runtime
arguments to the command.

This routine calls eval_pre to check the preconditions of the rule to determine if the rule can be executed at this time. If the precondition of the rule is satisfied, call_scheduler is called, which in turn will call the scheduler with the rule and its parameters.

If the precondition is false, eval_pre will return the specific predicate of the precondition that is false and place the runtime object that caused the predicate to be false in the correct symbol table entry in the rule. In this case try_back_chain() will be called to try and satisfy that predicate. If the effort succeeds, the activity is executed by calling call_scheduler(). Otherwise, a message is printed that the command cannot be executed and the routine returns.

Once the precondition is satisfied and the activity of the rule has been executed, the postcondition that corresponds to the return code of call_scheduler will be asserted by calling assert_posts(). If the assertion succeeds, do_forward_chain() is called to begin the forward chaining process.

### 10.7.4 try_back_chain()

```
int try_back_chain(rule)
     RULE_PTR rule;
```

rule -- the rule whose precondition is not satisfied

This routine tries to satisfy a precondition by backward chaining repeatedly until either the precondition is satisfied or there are no more alternatives for backward chaining. After each successful call to do_backward_chain, it calls eval_pre to check if backward chaining actually satisfied the failed precondition.

# Chapter 11

# Rule Overloading

The `overload` module contains code to handle rule overloading Overloading is a mechanism for implementing polimorphism in rules and objectes, i.e allowing multiple rules with same name, to operate on multiple objects with same name, in an object-oriented environment. Polymorphism involves disambiguating rule names and object names. Basically, the user requests a command $C$ with multiple arguments, where each argument is a name of an object. Since more than one rule can have the same name, there might be more than one rule whose name is $C$ that corresponds to the user command. Also, since objects do not have unique names in Marvel, each of the runtime arguments typed by the user at the command line can refer to more than one object that are instances of different classes. The `overload` module resolves the ambiguity by choosing the most appropriate rule to fire and the most applicable objects to pass ar paramters to the rule. The module operates in two modes that determine the ambiguity resolution mechanism: 1) the **DWIT** mode; and 2) the **DWIM** mode.

### 11.0.5 The DWIT Mode

**DWIT** the default mode and stands for *Do What I Tell.* In this mode the process of disambiguating the arguments passed from the command line, is seperated from the process of disambiguating the rule name.

First, the program disambiguates each argument by choosing the object that has the same name as the argument and that is the closest to the current object. For instance, if there are two different objects with the same name but different types (i.e., instances of different classes), the one that is closer to the current object (in the object hierarchy) will be chosen.

Then, the command name is disambiguated by choosing the rule whose parameters match the closest to the number, order, and classes of the runtime arguments that we have disambiguated in the first step. Notice that two types of 'closeness' are being used. In object resolution we use distance from current object in the objectbase hierarchy, whereas in rule resolution we use distance in the class hierarchy. The search order determines the distance. In object resolution the order is: 1. sub-tree of current-object (starting from itself); 2. all sub-trees of ancestors of current-object, starting from parent, grandfather and so forth, until the root object is reached. Within each level, the order is left-to-right.

3. Subtrees of other roots in the forest (left-to-right)

In rule resolution the order is variation of Breadth- first order, extended to handle multiple arguments, for details see documentaion of overload_rule(). I beleive that BFS is a better approach than DFS (used in CLOS) since it assures that the closest rule will be invoked, especially when multiple inheritance is used.

As for the measurment of the distance, notice that since we're dealing with sets of objects, the notion of distance means the minimal-vector, as shown in this example:

1. (1,2,2) ¡ (1,3,2)
2. (4,1,1) ¿ (1,4,4).

### 11.0.6 The DWIM Mode

DWIM stands for *Do What I Mean*. Unlike **DWIT**, in this mode the process of object resolution is affected by the available rules. For every candidate rule (i.e with same name) the objects are resolved with respect to the class type that is defined in the rule for the parameter. For instance, if r1[c1,c2] is rule r1, defined with two parameters of class c1 and c2 respectively, than resolving the call r1(o1,o2) from command line, will consider only objects with class types that are subclasses of c1 and c2 respectivelly. Then the closest object-set is chosen with its rule. If there are mutiple candidates still, and the object sets are identical in all candidates, than step 2 of **DWIT** is executed. Notice that if object sets are different, it means that we have two objects that are equi-distant from current object, so we can't disambiguate it and a message to the user is returned. So, part of the rule resolution process is done in the first phase together with object resolution. The advantages of this method are in resolving objects by a rule in some instances were DWIT can't find them.

OPEN QUESTIONS: 1. Is the overhead in first phase pays off by Eliminating the second phase sometimes ? 2. Is this extra resolution done is desired from the user's point of view ?

We now describe all the functions involved in overloading. They can all be found in the `overload` module, in `overload.c`.

### 11.0.7 chk_applicable()

```
int chk_applicable(rule,args,how_many)
     RULE_PTR rule;
     CMD_LINE_PTR args;
     int how_many;


rule -- the rule

args -- the command line arguments
```

how_many -- the number of arguments

This routine checks whether the rule passed to it can be applied on the given runtime arguments. It returns TRUE if the types and order of the parameters of the rule match the types and order of the runtime arguments.

Calls : External references:
is_class_or_subclass() (in evaluator/find_obj.c)

Called by: get_rule()

### 11.0.8 common_obj()

```
int common_obj(r1,r2)
     RULE_PTR r1,r2;
```

Tests if all objects in two rule's symbol tables are identical. returns TRUE if all objects are the same, FALSE otherwise.

Called by: find_min_rule_by_obj()

### 11.0.9 comp_rules()

```
int comp_rules(r1,r2)
     RULE_PTR r1,r2;
```

r1 -- the minimal rule so far

r2 - the current rule to be compared.

This routine compares the appropriatness of the two rules passed to it and returns 0 if the rules are equally appropriate; 1 if r1 is more appropriate; or 2 if r2 is more appropriate. The appropriateness of a rule is measured by the value of the BFS numbers assigned to each parameter of the rule. The BFS number measures how close (in the inheritance hierarchy) the type of a parameter is to the type of a runtime argument.

Called by: find_min_rule()

### 11.0.10 count_args()

```
int count_args(cmd_line)
     CMD_LINE_PTR cmd_line;
```

Count how many arguments there are in cmd_line.

### 11.0.11 cp_obj()

```
void   cp_obj(rule,args)
     RULE_PTR rule;
     CMD_LINE_PTR args;
```

Copies the objects from a rule's symbol table to the cmd_line.

Called by: resolve_objects()

### 11.0.12 dequeue()

```
int dequeue(queue)
     CLASS_Q_PTR queue;
```

Remove front element from queue

Called by: find_class_bfs()

### 11.0.13 enqueue()

```
int  enqueue(X,queue)
     CLASS_PTR X;
     CLASS_Q_PTR queue;
```

This routine inserts an item in a queue.

Called by: find_class_bfs()

### 11.0.14 find_candidates()

```
int find_candidates(arg,prules,how_many_args)
     char *arg;
     RULE_CHAIN_PTR *prules;
     int how_many_args;
```

arg -- rule name.

prules -- this is an output variable that gets linked list of
          all rules with same name.
          Notice that this is pointer to RULE_CHAIN_PTR !

how_many_args -- how many arguments were passed in the command line.

This routine traverse RuleTable and creates a list of all the rules
that have same name as arg.  prules will point to that list.

Returns: 1. the number of candidates found.
         2. The list of candidate rules.

Called by:  get_rule()

### 11.0.15 find_class_bfs()

```
int find_class_bfs(object_class,rule_class)
     CLASS_PTR object_class,rule_class;
```

object_class -- the class of a runtime argument.

rule_class -- the class of a rule parameter.

This routine performs breadth-first search on the class inheritance
hierarchy to determine the distance between the two classes passed
to it.  If the two classes are identical, then the distance is zero.

```
Calls:  enqueue()
        dequeue()
 free_queue()

Called by: overload_rule()
```

### 11.0.16 find_min_rule()

```
RULE_PTR  find_min_rule(possible_rules)
     RULE_CHAIN_PTR possible_rules;


rules -- all candidate rules with same name. (at least 2)

This routine finds the most appropriate rule to fire among the
rules found in the list of rules passed to the routinr.  The
appropriatness of firing a rule depends on the intellignece mode (DWIM
or DWIT).  The routine comp_rules() is called to compare the
appropriatness of two rules, and the most appropriate rule overall is
determined by repeated two-place comparisons.

Calls : comp_rules()

Called by: overload_rule()
```

### 11.0.17 find_min_rule_by_obj()

```
RULE_CHAIN_PTR  find_min_rule_by_obj(rules)
     RULE_CHAIN_PTR rules;


rules -- all candidate rules with same name. (at least 2)

This routine works more or less like find_min_obj. the difference
is that it may return more then one rule, since ties are acceptable,
in first phase of DWIM.

Calls : External references:
              make_struct()

        Internal references:
```

```
comp_rules()
common_obj()
free_chain()
```

Called by: resolve_objects();

### 11.0.18 find_objects()

```
int find_objects(args)
     CMD_LINE_PTR args;
```

args -- the list of arguments passed from the command line.

This routine traverses the argument list and replaces the names of objects in args with pointers to specific objects, whcih are found by calling find_obj(). If a pointer already exists in one of the entries of args, nothing is done. Notice that find_obj() is called with NULL as first parameter because no class restriction is imposed on find_obj() in DWIT mode.

Calls : External references:
find_obj() (find_obj.c)

Called by: get_rule()

### 11.0.19 find_objects_with_dist()

```
int find_objects_with_dist(rule,args)
     RULE_PTR rule;
     CMD_LINE_PTR args;
```

This routine is the DWIM version of find_objects().

Calls : External references:
find_obj_with_dist() (find_obj.c)

Called by: resolve_objects()

### 11.0.20 free_chain()

```
void free_chain(head)
     RULE_CHAIN_PTR head;
```

Free chain of rules.

### 11.0.21 free_queue()

```
void free_queue(head)
     Q_ELEMENT_PTR  head;
```

Free queue.

### 11.0.22 get_rule()

```
RULE_PTR  get_rule(rule_params,intelligence_flag)
     CMD_LINE_PTR  rule_params;
     int intelligence_flag ;
```

rule_params - A structure containig the name of the rule and the parameters from the command line. The parameters can be either names of objects (i.e., the names have to be disambiguated) or pointers to specific objects, if the user picks specific objects which is only possible in the graphics interface.

intelligence_flag - a marvel variable which determines the operation mode of the overload mechanism. This can be either DWIT (Do What I Tell you) or DWIM (Do What I Mean).

This routine finds the candidate rules (with same name and number of arguments) that correspond to a user command. The list of condidates depends on the intelligence mode, which determines the method of resolving ambiguity. The routine first calls find_candidtes() to get all the rules whose name and whose number of paramters is the same as the name of the command and number of arguments passed. It then calls find_objects to disambiguate the command arguments that are names of objects rather than pointers to objects. If the argument passed in rule_params is a pointer to a specific object (i.e., chosen by clicking on an object in the graphics interface -- this is still unimplemented), there is nothing to

disambiguate.

Finally, from the list of candidate rules, the routine resolve_objects() and chk_aplicable() to choose the rule that could correctly implement the user command given the runtime arguments that are passed.

The routine returns a pointer to the chosen rule. As a side effect, it fills in the rule_params structure with the pointers to objects rather than names of objects after disambiguating the names.

Calls: Internal references:
chk_applicable()
find_candidates()
find_objects()
overload_rule()
resolve_objects()

Called by:

expand_and_execute_command() (in interpreter/ci.c)

### 11.0.23 overload_rule()

```
RULE_PTR  overload_rule(possible_rules,args)
     RULE_CHAIN_PTR possible_rules;
     CMD_LINE_PTR args;
```

possible_rule -- list of all applicable rules

args -- list of runtime arguments to the user command.

This function chooses one of the rules from the list passed to it depending on the types of the arguments. The approach taken is Breadth-first search. A number, called a BFS number, is assigned to each parameter of each rule in the list. The BFS number indicates the distance between the type of the runtime object in args and the type of the corresponding paramter of the rule. If the two types are identical, then the distance is zero. If the type of the runtime object is a first ancestor (in the multiple inheritance hierarchy) of the type of the rule's parameter, then the distance is 1, and so on. If the type of one of the runtime objects and the type of the corresponding rule parameter are incompatible (i.e., there is no inheritance relationship between those two), a special value, CLASS_NOT_FOUND, is assigned. After assigning the BFS numbers for

each rule, find_min_rule() is called to determine which is the most appropriate rule.

Called by: get_rule()

### 11.0.24 resolve_objects()

```
RULE_CHAIN_PTR resolve_objects(rules,args)
     RULE_CHAIN_PTR rules;
     CMD_LINE_PTR args;
```

rules -- list of rules that can potentially be chosen.

args -- list of arguments that will be used to choose appropriate rule.

This routine is used when DWIM mode is on. DWIM mode forces the disambiguation of object names not only through the types of objects, but also using the distance between the objects and the current object. This is based on the assumption that when the user is working on a current object, he intends the commands that he invokes to operate on the current object and other objects in its vicinity. Thus, the distance between an object and the current object becomes a factor in choosing which rule to fire in order to implement the user command.

Calls : External references:
find_min_rule_by_obj
find_objects_with_dist

Called by: get_rule()

# Chapter 12

# The Marvelizer

This chapter discusses the Marvelizer.

## 12.1 Basic Marvelizing

Found in marvelize.c.

### 12.1.1 cleanup_marvelize()

```
void cleanup_marvelize(s1, s2)
char *s1, *s2;
```

```
Cleanup any allocated stuff from the marvelizer.
Cleanup the postfixes array also.
Reset the verbose flag back to it's original value.

s1 and s2 are regular character strings.
```

### 12.1.2 do_marvelize()

```
int do_marvelize(class, root)
CLASS_PTR class;
char *root;
```

```
So it seems that we have all the information we need to begin some
serious marvelizing.  Here's the algorithim:

1.  Create a new instance of the passed in class for the passed in root.
```

```
   This instance must be the root of an entirely seperate db tree. It
   must have nothing common with the tree in memory.

2. Use readdir to search the directory.
   If the entry is:

   a directory, if there is only one large attribute, use it to make
      a new child instance, and link to parent.  Recurse this step with
      the contents of the found directories.

      if there are multiple large attributes, use the class,
      instance list information to decide where it goes.  If
      not listed and verbose or quiet mode, query user about where
      to put the beast.  If silent, put in first large attributes
      instance list.

   a reg. file if it's postfix matches one in the classes postfix list,
      create an extra directory for it, put the file there, and link in
      this link instance.

   anything else, just leave it be.

3. Now we have a whole objectbase tree starting at root, we need to
   finish linking the sucker in.  Starting with the root, do normal
   tree recursion (like in printing module).  At each instance, call
   the global linking routine, and all the other stuff that
   addinstance would do.  Note that there should be no freeing which
   needs to take place.

4. return with succes if this be the case.

class -- the class at which to start marvelizing the object given.
         this class has been verified to exist.
root -- the location in the file system if the project to be marvelized.
        this location has been verified to be valid.
```

### 12.1.3 do_marvelize_recurse()

```
int do_marvelize_recurse(class, root, parent_inst)
CLASS_PTR class;
char *root;
INSTANCE_PTR parent_inst;


Recursively go down the tree started at root, making objects and adding
files as appropriate.  The real tough work of matching against the
```

class_info list is done in the match_...() routines.

Note that treatment of files which have a match in the pfx table, and that match corresponds to a class. In this case, we create an object with the name of the file in in, and then copy the physical file into that object (directory).

We use set_current_inst() here to keep on changing the system current object, but when we back track from recursion, the current object will then always be at the incorrect level. So before backtracking, remember to reset the current object to the next level of hierarchy.

In these routines, I have tried to be clear about which level the various classes and objects are, but it is complex, and thus will doubtless appear unclear.

class -- the class to which the parent_inst belongs.
root -- the path on the file system to the current root being marvelized.
parent_inst -- the parent object which was just newly created by the
previous level of recursion.

### 12.1.4 initialize_marvelize()

```
void initialize_marvelize()
```

Initialize any goodies for the marvelizer. Currently, this is just the global postfixes array. Be certain to temporarily turn off the verbose flag, to avoid having to say yes to adding a zillion objects.

### 12.1.5 marvelize_cmd()

```
/*ARGSUSED*/
int marvelize_cmd(cmd_line)
CMD_LINE_PTR cmd_line;
```

This command asks a series of questions to gather all the data it needs to marvelize somewhat arbitrary bits of code. The concept here is that it can be done a directory at a time, or in any other convienent package.

### 12.1.6 match_file_postfix()

```
int match_file_postfix(class, name)
CLASS_PTR class;
char *name;
```

This routine searches the class_info table for a class matching the passed in class, and with a matching postfix to name. Specifically:

1. Use given class to find corresponding entry in class_info. This entry will match class pointer AND be of type CI_PFX.

2. Do postfix_strcmp() on each of the postfixes under class. If a match is found, return TRUE.

3. If one or two above fail, return FALSE.

class -- the class in question.
name -- the object name, to be compared against the postfixes.

### 12.1.7 match_large_att_class_info()

```
ATTRIBUTE_PTR match_large_att_class_info(class, name)
CLASS_PTR class;
char *name;
```

class -- the owner class of the desired object.
name -- the name of the desired object.

This somewhat complex routine searches the class_info table for a class in which to put the given name. Specifically:

1. Use the passed in class to get a list of large attributes.

2. If there is only one large attribute, return it.
   Otherwise, search for classes coresponding to the large_atts->d_name, AND classes whose type is CI_MULTI.

3. Search for name amongst one of those classes. Return the first one found. This is only ambiguous if the user inputs are bogus.

4. If no name is found, return NULL, signifying that a query must take

place (see resolve_with_query()).

### 12.1.8 match_large_att_class_pfx()

```
ATTRIBUTE_PTR match_large_att_class_pfx(class, name)
CLASS_PTR class;
char *name;
```

This routine finds the first occurence of name which matches a pfx in the table, where the corresponding class in the found pfx matches one of class's large_att->dname fields.

class -- the owner class of the desired object.
name -- the name of the desired object.

### 12.1.9 postfix_strcmp()

```
int postfix_strcmp(str, pfx)
char *str, *pfx;
```

str -- the string
pfx -- the postfix. Note that it is legal for the postfix to be longer then the string, in that case FALSE will be returned.

Compare two strings, starting at the back.

Return TRUE is the second string is a postfix of the first, FALSE otherwise.

### 12.1.10 process_class_lists()

```
int process_class_lists(valid, string, type)
int *valid;
char *string;
int type;
```

Process each input line, which tells various things to do with particular classes in the system.

```
cl_indx is the number in the class_list array to put this dude.

valid should be set to TRUE or FALSE, depending upon whether
the this string is valid input.

string is the string to process.
```

## 12.2 Advanced Functionality

Found in m_query.c.

### 12.2.1 create_query_file()

```
void create_query_file(fp, class)
FILE *fp;
CLASS_PTR class;
```

### 12.2.2 edit_marvelize_queries()

```
void edit_marvelize_queries(file)
char *file;
```

```
This routine edits the temporary file generated by get_marvelize_query().
The editing is done with the users favcrite editor, with a default of
vi.
```

### 12.2.3 execute_marvelize_queries()

```
FILE *execute_marvelize_queries(file)
char *file;
```

```
This routine executes the queries on the originating system specified by
the marvelize_cmd() routine.  It returns a file pointer to the open
file of output from the command, waiting to be shoved back into
marvel.
```

### 12.2.4 get_marvelize_queries()

```
char *get_marvelize_queries()
```

This routine searches through the objectbase and generates a incomplete list of queries. It writes this list to a temporary file, and returns a pointer to the name of this file.

### 12.2.5 stuff_back_values()

```
/*ARGSUSED*/
void stuff_back_values(orig_file, fp)
char *orig_file;
FILE *fp;
```

This routine takes the values created by the program, casts them appropriately for the objectbase, and stuffs them in the appropriate places.

# Chapter 13

# The Marvel Executable

The main MARVEL program utilizes the `shared.a` library, and a main routine found in `marvel.a`. Thus there is not much meat left to put here.

## 13.1 The Main Loader Program

The following routine is from the file `main.c`.

### 13.1.1 main()

```
main(argc, argv)
int argc;
char *argv[];
```

The main routine for the marvel executable.

# Chapter 14

# The Loader Executable – Semantics

This chapter discusses the semantics module of the external Loader program. The Loader is a seperate process for historical reasons at this point. It uses almost all of the code in the shared MARVEL library.

This module contains all the routines that perform the semantics checking and compilation of forward and backward chains when loading an MSL file. Following, we describe each one of these routines. The code of all of them can be found in the semantics module.

## 14.1 Compiling the forward and backward chains

The following routines are from the file chk_chains.c.

### 14.1.1 add_pointer()

```
CHAINS_PTR add_pointer(chain, predicate)
     CHAINS_PTR chain;
     PRED_TABLE_PTR predicate;


chain -- pointer to a chain that represents either a forward
         chain or a backward chain.

predicate -- the predicate that should be attached to the
             chain

This routine adds the rules that include the predicate in their
precondition or postcondition to the chain (representing either a
forward chain or a backward chain).  The predicate might be included
in several rules;  This routine only adds the rules that are not
already on the chain.
```

### 14.1.2 chk_possible_backward_chains()

```
void chk_possible_backward_chains(pred_entry)
     PRED_TABLE_PTR pred_entry;
```

```
pred_entry --  the entry into the pred table in question.

This routine checks the left side and the right side of a
precondition (pred_entry) with the left side of every postcondition in
the pred_table. Only ATTRIBUTES are allowed for both preconditions and
postconditions. If either sides of the precondition match a
postcondition then a list of backward pointers of the rules that include
the postcondition will be attached to the backward chaining pointer of
the pred_entry.
```

### 14.1.3 chk_possible_forward_chains()

```
void chk_possible_forward_chains(pred_entry)
     PRED_TABLE_PTR pred_entry;
```

```
pred_entry --  the entry into the pred table in question.

This routine compares the pred_entry passed to it with every other
predicate in PredTable.  If a predicate that can be forward chained to
is found, the rules that include that predicate are added to the
forward list of the pred_entry.  Duplictae rules on the same list are
not allowed.
```

## 14.2 Collapsing complex conditions

The following routines are from the file collapse.c.

### 14.2.1 collapse()

```
void collapse(condition)
     COND_PTR condition;
```

```
condition -- a condition to collapse
```

```
This function collapses a tree so that duplicated operators are
reduced to single operators. For instance, (AND (AND .. ).. ) == AND
It is a modified breadth first traversal, with depth lookahead for
comparison of operators.  The syntax does not allow uncollapsed
postconditions, and this routine will potentially make a mess of
postconditions.  So only use it for the various kinds of preconditions.
```

### 14.2.2 collapse_bindings()

```
void collapse_bindings(bindings)
     BINDING_PTR bindings;


bindings -- a pointer to the bindings.

Just like collapse(), but does it for each characteristic in each part
of the binding.
```

### 14.2.3 compare_predicates()

```
static int compare_predicates(i1, i2)
     int i1, i2;


i1, i2 -- the index's of the predicates

Compare two predicates, to determine if they are identical.  Compare all
the parts seperately.  TRUE means they are identical.
```

### 14.2.4 compare_tree()

```
static int compare_tree(top1, top2)
     COND_PTR top1, top2;


top1 - the top of a tree

top2 - the top of a tree
```

This function compares 2 trees to determine whether they are identical. The trees are traversed preorder with each node being matched for like operators and predicates. A TRUE return value indicates that the two trees are identical.

### 14.2.5 do_collapse()

```
void do_collapse(parent, children)
     COND_PTR parent, children;
```

parent --

children --

This routine collapse a cond tree.

### 14.2.6 free_cond_tree()

```
void free_cond_tree(cond, top)
     COND_PTR cond;
     int top;
```

cond -- the condition to be freed

top --

Free a condition, by searching down the child for all conditions, and freeing from the bottom up. Make sure not to free any next pointers at the top level.

### 14.2.7 look_up_var()

```
SYMBOL_PTR look_up_var(variable, symbol)
     char *variable;
     SYMBOL_PTR symbol;
```

variable - a variable (ie ?p)

```
symbol - the symbol table containing variable entries

This function searches a variable list to determine whether a
variable exists in the list. If it is defined, the variable entry is
returned, otherwise a NULL pointer is returned.
```

### 14.2.8 remove_dups()

```
void remove_dups(top)
     COND_PTR top;


top -- the tree representiong a condition;

This function removes duplicate portions of a tree. It checks for
both identical predicates within an operator and identical subtrees with
in the tree.

For postconditions, this should be called for each seperate condition,
rather then all of them.
```

### 14.2.9 remove_dups_bindings()

```
void remove_dups_bindings(bindings)
     BINDING_PTR bindings;


bindings -- a pointer to the bindings.

Just like remove_dups(), but does it for each characteristic in each
part of the binding.
```

## 14.3 Looking Up Information

The following routines are from the file `l_lookup.c`.

### 14.3.1 find_rule_in_chains()

```
CHAINS_PTR find_rule_in_chains(rulelist, rulename)
     CHAINS_PTR rulelist;
     char *rulename;

    rulelist - list of pointers to entries in the activity
               activity table.

    rulename - name of rule

    This routine searches forward or backward pointers to determine
    whether a pointer to an entry in the activity with the specified
    rule name exists.
```

### 14.3.2 find_strat_name_in_strlist()

```
STRLIST_PTR find_strat_name_in_strlist(strategy, strat_list)
     char *strategy;
     STRLIST *strat_list;

    strategy - the name of the strategy

    strat_list - list of strategies that is searched

    This function searches a list of strategies for a strategy. If the
    strategy is found, a pointer to the entry in the list is returned. Other-
    wise, the NULL pointer is returned.
```

## 14.4 The Loading Routines

The following routines are from the file load_guts.c.

### 14.4.1 build_binding_symbols()

```
void build_binding_symbols(rule)
     RULE_PTR rule;

Find all the given rule's bindings and build a symbol (if there was
not already one there) for each bound variable.  There should have not
been a symbol at this point.  The class of the bound variable has
already been checked in the parser for existence.
```

### 14.4.2 build_rule_table()

```
void build_rule_table(name, params, bindings,
                      prop_list, actions, posts)
     char *name;
     SYMBOL_PTR params;
     BINDING_PTR bindings;
     COND_PTR prop_list, posts;
     ACTLIST_PTR actions;
```

name -- the name of the rule

params -- the paramter list of the rule

bindings -- the bindings of the preconditions of the rule

prop_list -- the property list of the preconditions of the rule

actions -- the activity list of the rule

posts -- the postconditions of the rule.

This routine calls the functions to create a rule by linking all the structures passed to the routine to the rule structure, and then adding the rule to the RuleTable. Some of the fields in the predicates of the characteristic function, precondition and postconditions of the rule, which are inserted in the PredTable by the parser, are filled in.

### 14.4.3 check_postcondition_variables()

```
void check_postcondition_variables(rule, conds)
     RULE_PTR rule;
     COND_PTR conds;
```

Search through all the postconditions of a rule, checking to be sure that any variables are parameter type variables rather then bindings. This assures some semantic consistency in the rule's postconditions.

### 14.4.4 create_rule()

```
RULE_PTR create_rule(rule_name, params, activities, bindings,
                                prop_list, post_conds)
     char *rule_name;
     SYMBOL_PTR params;
     ACTLIST_PTR activities;
     BINDING_PTR bindings;
     COND_PTR prop_list, post_conds;
```

This function creates a RULE_PTR structure and inserts it in the RuleTable if an equivalent rule doesn't already exist in the table. If an equivalent rule already exists, all information will be merged into the existing rule, by calling merge_conditions for the characteristic function, precondition, and postcondition and merge_variables for the symbol table. In this case, the routine will return a pointer to the existing rule after the information has been merged.

Two rules are equivalent if they have the same name, the same parameter list (i.e., the types and order of all their paramters are the same) and the same activities. If the new rule has the same name as a rule in the RuleTable, but the activities differ, the new rule is ignored.

### 14.4.5 handle_acts()

```
int handle_acts(activities, variables)
     ACTLIST_PTR activities;
     SYMBOL_PTR variables;
```

Make sure there are symbols for all the arguments in an activity. If not, return an appropriate error.

### 14.4.6 process_bindings()

```
void process_bindings(bindings, symbols)
     BINDING_PTR bindings;
     SYMBOL_PTR symbols;
```

The bindings contain seperate characteristics for each binding. So handle them seperately.

### 14.4.7 process_conditions()

```
void process_conditions(conds)
     COND_PTR conds;
```

conds -- the condition which will be processed.

This routine goes through all the predicates of the conditions passed to it, and for each predicate it fills in the strategy field (i.e., the strategy to which the predicate belongs) with a pointer to the current strategy being loaded.

## 14.5 The Main Loader Program

The following routines are from the file main.c.

### 14.5.1 main()

```
main(argc, argv)
int argc;
char *argv[];
```

This is the main routine for the loader executable.

### 14.5.2 marvel_sig_bus()

```
int marvel_sig_bus()
```

Set the bus error signal handler for the loader.

### 14.5.3 marvel_sig_fpe()

```
int marvel_sig_fpe()
```

Set the floating point violation signal handler for the loader.

### 14.5.4 marvel_sig_segv()

```
int marvel_sig_segv()
```

Set the segmentation violation signal handler for the loader.

### 14.5.5 set_temp_filename()

```
int set_temp_filename()
```

Sets up the global string tempfname[] with the name of a file that can be used for temporary storage. Each procedure that uses this variable is responsible for checking to make sure the file can be created there (i.e. the user may have removed that directory, write protected it, etc... It finds the correct directory according to these steps:

1) check the local directory for write access.
2) check $TMPDIR in envrionment.
3) /tmp

Returns TRUE if valid directory found, otherwise FALSE.

## 14.6 Merging Strategies

The following routines are from the file merge.c.

### 14.6.1 merge_bindings()

```
BINDING_PTR merge_bindings(old_bindings, new_bindings)
     BINDING_PTR old_bindings, new_bindings;
```

old_bindings - old bindings

new_bindings - new ones

### 14.6.2 merge_conditions()

```
COND_PTR merge_conditions(old_conditions, new_conditions, cond_type)
     COND_PTR old_conditions, new_conditions;
     int cond_type;


old_conditions - old u_quants, char_funcs, prop_list or post
                 conditions

new_conditions - new ones

cond_type - flag indicating the kind of condition
This routine merges two trees together based on the types of the
conditions. For pre_conditions and characteristic functions, the two
trees are AND'ed together. For post conditions, the postconditions are
XOR'ed together. The merged tree is returned.
```

### 14.6.3 merge_variables()

```
void merge_variables(rule, newvars)
     RULE_PTR rule;
     SYMBOL_PTR newvars;


rule -- this rule

newvars -- it's current symbol table

This  function merges the variables defined in a strategy into
a symbol table for a rule that exists in the rule table. Each
variable in a rule is unique. If a variable is already defined in the
symbol table, a new name is assigned to any variable with the same name
that is being added to the list.
```

## 14.7 Ordering the Loading of Imported Strategies

The following routines are from the file oreder_imps.c.

### 14.7.1 order_imports()

```
STRLIST_PTR order_imports(strategy_name)
     char *strategy_name;

    strategy_name -- the name of the strategy to start from

    This routine collects a list of strategies in a proper order for loading
    or merging.
```

## 14.8 The Parser Routines

The following routines are from the file parse_msl.c.

### 14.8.1 compile_chain_network()

```
compile_chain_network()
```

### 14.8.2 do_parse_msl()

```
int do_parse_msl(cmd, full_strategy_name)
int cmd;
char *full_strategy_name;

This is the heart of load, merge and unload.  Do the thang, and if there
is ever an exit detected, immeadiately leave, in order to waste as little
time as possible.
```

### 14.8.3 get_all_strategies()

```
int get_all_strategies(strategy_list)
STRLIST_PTR strategy_list;
```

### 14.8.4 link_cond_to_rule()

```
link_cond_to_rule(cond, rule)
   COND_PTR cond;
   RULE_PTR rule;


cond -- condition
rule -- rule to which the condition belongs

Link up all conditions of a rule to the rule, so the rule can be found
from any condition.
```

### 14.8.5 link_pred_to_rules()

```
link_pred_to_rules()


Go through all the rules in the rule table, and for each one, link
all the predicates in all the bindings, property list, and
postconditions.

Make sure to check all the bindings, not just the first.
```

### 14.8.6 make_and_open_msl_file_name()

```
static int make_and_open_msl_file_name(file)
char *file;
```

### 14.8.7 make_strategy_path_name()

```
int make_strategy_path_name(location, name)
char location[];
char *name;
```

### 14.8.8 run_parser()

```
int run_parser(file)
char *file;
```

### 14.8.9 yyerror()

```
yyerror(s)
char *s;
```

```
Print out an error about the nature of the parser error just found.
Set a flag to remember that at least one syntax error has been found
(SyntaxError).
```

## 14.9 Loading Relations

The following routines are from the file `relation.c`.

### 14.9.1 build_rel_table()

```
void build_rel_table(name, domain, range)
     char *name, *domain, *range;

    name - relation name

    domain - domain of the relation

    range - range of relation

    This function adds information to the relation table
```

### 14.9.2 make_rel()

```
REL_TABLE *make_rel(name, dom, ran)
     char *name, *dom, *ran;

    name - relation name
```

```
domain - domain of the relation

range - range of relation

This routine allocates space for an entry in the  relation table
and copies the information  into it.
```

## 14.10 Resetting The Lexer

The following routines are from the file `reset_lex.c`.

### 14.10.1 reset_lex()

```
void reset_lex()
```

# Chapter 15

# The Loader Executable – Parsing

This chapter discusses the `parser` module of the external Loader program. The Loader is a seperate process for historical reasons at this point. It uses almost all of the code in the shared MARVEL library.

This module contains the `yacc` and `lex` files that constitute the MSL parser. Following, we describe each one of these files. The code of all of them can be found in the `parser` module.

# Chapter 16

# Reading and Writing the State of the System

The load module contains functions to read and write the state of the MARVEL objectbase, data and process models at any time. It is used to create savepoints, including the savepoints that represent temporary exits from the system.

All this information is kept in a set of ascii files in the data directory of each MARVEL objectbase. Of particular importance are the files objectbase and strategy, these files represent the state of the system after the last quit, and allow things to be restarted in an identical fashion at some later time. objectbase contains information pretaining to all the current objects, and strategy contains the currently loaded data and process models. All the other files represent savepoints of these two files. Such savepoints are currently done rather frequently (in an automatic fashion), due to the state of the system, however this is likely to change as *Marvel* becomes more and more stable.

These files are all ascii files, to aid in debugging the evolving system. We are aware of the efficiency tradeoffs of making these files binary, and believe it is worth leaving as they are for the time being.

## 16.1 Reading the State

The code for reading the state of a MARVEL OBJECTBASE is in the files loader.c and r_state.c.

### 16.1.1 read_objectbase()

```
int read_objectbase()
```

```
This routine reads in the image of the objectbase that was stored in the
previous invocation of Marvel.  The image is stored in the data
directory of the Marvel database.
```

### 16.1.2 read_strategies()

```
int read_strategies(str_f, merge)
     char *str_f;
     int merge;
```

This routine reads in the strategies that were stored in the previous invocation of Marvel.

### 16.1.3 read_objbase()

```
int read_objbase(fp)
     FILE *fp;
```

fp -- file pointer to the open file containing all the intermediate form structures which get read.

This routine reads in all the instances of the objectbase. The class structure should have already been read in at this point.

### 16.1.4 read_classes()

```
void read_classes(fp, merge)
     FILE *fp;
     int merge;
```

fp -- file pointer to db file.

merge -- true if merging classes, false if loading in a new set.

This routine reads in all the new classes from the temporary file. Do this by first marking all existing classes active flag FALSE. Then create an entire new list. Then, if merging, join the new with the old, with common defaults superceeding those of the old. If loading, keep the old physical class, but put in all the new attributes, removing any of the old ones not currently used.

### 16.1.5 read_all_classes()

```
CLASS_PTR read_all_classes(fp)
     FILE *fp;

    fp -- file pointer to db file.
```

### 16.1.6 read_rule_table()

```
RULE_PTR read_rule_table(fp)
     FILE *fp;
```

This function reads in the entries of the activity_table in order to reconstruct the table. This function returns the front of the reconstructed table. The pre and post trees are not read in by this function.

### 16.1.7 read_conditions()

```
COND_PTR read_conditions(fp, root_cond)
     FILE *fp;
     COND_PTR root_cond;
```

This routine reads in a string of numbers, and turns them into arbitrarily complex nested cond structures. This is all done recursively

A key for the numbers is as follows:

-1 AND
-2 OR
-3 NOT
-4 end of this cond.
<other number> index into the pred table, to find the real cond.

thus -2 -1 40 41 -4 42 -4 -4

would be

```
(or (and (40) (41))
    (42))

this could be interpreted as two postconditions, or a complex
precondition.

if NULL is passed in as root_cond, this signals the begining of
a condition.  It is assumed that the string will be correct, so no
error checking is done.

A similar examples is found under write_condtions().
```

### 16.1.8 read_pre_post()

```
void read_pre_post(fp)
     FILE *fp;

    fp -- the file pointer into the strategy file that is now pointing
          at a list of conditions.
```

### 16.1.9 read_pred_table()

```
int read_pred_table(fp)
     FILE *fp;

   This routine fills up the global PredTable with the entries in the file
   that fp points to.  It is assumed that the entries in the file are in
   the correct numerical order.

   TRUE is returned if all is fine, FALSE if there is a problem suggesting
   that
```

### 16.1.10 read_rel_table()

```
REL_TABLE_PTR read_rel_table(fp)
     FILE *fp;

    fp -- pointer to file containing info about relations.
```

This function reconstructs the relation table. It reads in each entry from the file specified in the filename. The front of the table is is returned.

### 16.1.11 read_strat_table()

```
STRATEGY_PTR read_strat_table(fp)
     FILE *fp;
```

fp -- file pointer to the db file.

This routine loads the strategy structure from the file.

### 16.1.12 set_subclasses()

```
void set_subclasses(root_class)
     CLASS_PTR root_class;
```

root_class -- the head of the list of classes.

This routines goes through the list of classes and sets appropriate sub classes for everything, based upon the superclasses currently present. Note special treatment for the two classes TOOL and ENTITY, which are the only two who have a NULL address field in the superclass record. These are just skipped, rather then wasting the time to create a dummy subclass.

### 16.1.13 add_subclass()

```
void add_subclass(class, sub)
     CLASS_PTR class, sub;
```

class -- the class which gets the subclass.

sub -- the class to point the new subclass at.

add a subclass corresponding to a superclass of some other class.

### 16.1.14 get_next_att_tag()

```
int get_next_att_tag()
```

### 16.1.15 link_owner_att()

```
static void link_owner_att(inst)
     INSTANCE_PTR inst;
```

The idea here is to link up the inst->owner_att field. Not as easily said as done. The inst record has the following fields:

```
     own_att_tag
     own_att_mclass
```

note that this inst's owner_class will be the same as the owner_att's class. If own_att_mclass is NULL (a top level inst), do nothing. otherwise, search down own_att_mclass's global list of instances for an appropriate attribute whose tag matches own_att_tag.

### 16.1.16 make_large_att_info()

```
ATTRIBUTE_PTR make_large_att_info(fp, attname, b1, b2)
     FILE *fp;
     char *attname;
     int b1, b2;
```

### 16.1.17 make_medium_att_info()

```
ATTRIBUTE_PTR make_medium_att_info(attname, b1, b2)
     char *attname;
     int b1, b2;
```

## 16.2 Writing the State

The code for writing the state of a MARVEL OBJECTBASE is in the files `loader.c` and `w_state.c`.

### 16.2.1 write_objectbase()

```
void write_objectbase()
```

This routine dumps an image of the objectbase onto a file and stores the file in the data subdirectory of the project database. The routine creates a unique name for the file.

### 16.2.2 write_strategies()

```
int write_strategies(str_f)
     char *str_f;
```

### 16.2.3 write_classes()

```
void write_classes(fp)
FILE *fp;
```

Write out information about the classes. First write if their are none. If there are, first write out a list of all the classes, to aid reading in later. Than for each class, write it's super classes, then all about it's small, medium, then large attributes.

### 16.2.4 write_conditions()

```
void write_conditions(fp, conds)
```

```
FILE *fp;
COND_PTR conds;
```

This routine writes out a condition. It does a recursive traversal of the cond, writing out a prefix notation sort of condition that will be easy to regenerate upon rereading time. The following codes are used torepresent the connectors:

```
-1 -- AND
-2 -- OR
-3 -- NOT
-4 -- end of some corresponding connector.
```

Predicates are represented by numbers >= 0, which correspond to the index of the actual predicate in the system PredTable.

Thus, a condition like

```
(not (and a b (or c (and g h) d b (and d f g))))
```

is written out as:

```
-3 -1 a b -2 c -1 g h -4 d b -1 d f g -4 -4 -4 -4
```

with all the letters replaced by the appropriate indicies of PredTable. there is not punctuation written, so it is very easy to read these expressions back in.

A similar example is to be found for read_conditions, if this is not clear.

### 16.2.5 write_objbase()

```
void write_objbase(fp)
FILE *fp;
```

Write out the objectbase. That is, all about the objects, not about the classes.

For each class, write the name and then all the instances of that class. Each instance gets its link tags, all the information about it's attributes, first small, remembering to get all the attribute link info, then medium, then large.

### 16.2.6 write_pre_post()

```
void write_pre_post(fp)
FILE *fp;
```

Write about all the pre and post conditions in the system. This routine captures the structure of pre and post-conditions, as compared to the write_pred_table() routine, that fully describes each predicate.

For each rule currently loaded, write the following:

** It's parameters
** It's bindings, including quantifiers and characteristics
** It's property list
** It's postconditions, expressed as an OR type condition.

Write conditions is called ot do all this writing.

### 16.2.7 write_pred_table()

```
void write_pred_table(fp)
FILE *fp;
```

Write out the predicate table. If it is empty, just write that. otherwise, for each predicate do the following:

write out the predicate type, the type of the second operand, the operator and the string representing the first operand, then the representation for the second operand, which might be one of several different types. Then skip a line.
Next, write out the strategies in which this predicate is found, the forward chains, the back chains, and the owner rule. These all go on seperate lines. Lastly, write out the parameters to the rule, so the proper rule can be found in case of overloading.

### 16.2.8 write_rel_table()

```
void write_rel_table(fp)
FILE *fp;
```

```
This routine writes out the entries of the relation table into a file
All the information associated with a relation is written out so that
the table can be reconstructed at some later time. Note that the rel-
ation information is quite bogus right now.
```

### 16.2.9 write_rule_table()

```
void write_rule_table(fp)
FILE *fp;


Write out information about the rules currently loaded in the system.
If there are none, write that, otherwise print out the name, how many
parameters there are, the rules symbol table information, the rule's
activities, and all the strategies the definition comes from.

Symbol table information includes the name, type of symbol, which
parameter in the list and the name of the class that the symbol repre-
sents.
```

### 16.2.10 write_strat_table()

```
void write_strat_table(fp)
FILE *fp;


Write out all the strategies in the system, and each strategy's import
and export (not used) list.
```

A few miscellaneous related routines in `loader.c` follow.

### 16.2.11 copy_attrib()

```
ATTRIBUTE_PTR copy_attrib(atlist,att)
     ATTRIBUTE_PTR atlist, att;


atlist -- list of attributes

att -- the attributes we want to add to the list.
```

```
This routine creates a new attribute structure, copies the information
from att into it, and attaches the new structure to the attribute list
(attlist).  It returns the updated list.
```

### 16.2.12 fre_strategies()

```
void fre_strategies()


This routine frees up all the structures that were allocated in order
to load the various parts of the strategies that are in memory.  This
includes freeing up the RuleTable,  the PredTable, the relations table,
and the strategy table. Currently this routine does not do an
exhaustive freeing job, rather it leaves some junk memory behind.
```

## 16.3 Merging the Data Model

The new data model must be merged with the existing one if the user just performed a merge command. This is done with the following routines, found in obj_merge.c.

### 16.3.1 fix_class_hierarchy()

```
void fix_class_hierarchy(new_root, merge)
CLASS_PTR new_root;
int merge;


Adds the list of classes to the existing object base.
If an object is already defined in the object base, the information is
merged into the existing structure. Otherwise, the class is added to
front of the class list (object base).

new_root -- list of classes to be loaded/merged into the object base
merge  -- whether to merge or load the new stuff.
```

### 16.3.2 fix_class()

```
static void fix_class(old, new, merge)
```

```
CLASS_PTR old, new;
int merge;


either merge or load this particular class.  Separate out the work into
the superclasses, the attributes, and the instances.

old -- the already existing (out of date) class
new -- the new class
merge -- whether or not to merge
```

### 16.3.3 fix_atts()

```
static void fix_atts(old, new, merge)
CLASS_PTR old, new;
int merge;
```

### 16.3.4 fix_instances()

```
/*ARGSUSED*/
static void fix_instances(old, new, merge)
CLASS_PTR old, new;
```

### 16.3.5 find_subsuper()

```
SUBSUPER_PTR find_subsuper(name, address, sub_sup_list)
char *name;
CLASS_PTR address;
SUBSUPER_PTR sub_sup_list;


this routine finds a subclass or superclass entry.  It searches for
either the name or address, one or the other should be NULL.

name -- the name to look for
address --  the class to look for
```

```
sub_sup_list -- the list of sub or superclasses to look in.
```

### 16.3.6 fix_supers()

```
static void fix_supers(old, new, merge)
CLASS_PTR old, new;
int merge;


merge or load the superclasses of the new class with
those of the old class.  old is the one that prevails if there
are questions.  For a load, just replace the super list with the
existing one.  For a merge, actually merge them in.  Note that
subclasses will be recalculated after all this is complete.

old -- the original existing class
new -- the newcomer being merged/loaded
merge -- TRUE if merging, FALSE is loading
```

## 16.4 Adding Predicates to the Predicate Table

Does this stuff not belong elsewhere?? File add_pred.c

### 16.4.1 add_int_pred_to_table()

```
int add_int_pred_to_table(op1, op2, operator)
char *op1;
int op2, operator;


Make a predicate table entry, bumping the size while doing it.  This
routine does not currently check for duplicate entries.  Return the
index of the new entry, or -1 for failure.
```

### 16.4.2 add_pred_to_table()

```
int add_pred_to_table(op1, op2, operator)
char *op1, *op2;
```

```
int operator;
```

Make a predicate table entry, bumping the size while doing it. This routine does not currently check for duplicate entries. Return the index of the new entry, or -1 for failure.

### 16.4.3 add_real_pred_to_table()

```
int add_real_pred_to_table(op1, op2, operator)
char *op1;
double op2;
int operator;
```

Make a predicate table entry, bumping the size while doing it. This routine does not currently check for duplicate entries. Return the index of the new entry, or -1 for failure.

# Chapter 17

# Miscellaneous Functions

The module misc contains many miscellaneous routines. These should probably be moved to proper locations.

## 17.1 Functions

### 17.1.1 compare_acts()

```
int compare_acts(old_activity, new_activity)
     ACTLIST_PTR old_activity, new_activity;

   old_activity - activity list of a rule in the rule table

   new_activity - activity list of new rule  (with the same name)

   Compare the two activities.  Return TRUE if they are the same, or FALSE
   otherwise.  The comparison includes the tool, the operation and all the
   types of arguments of the activity.
```

### 17.1.2 compare_args()

```
int compare_args(old_symbols,new_symbols)
     SYMBOL_PTR old_symbols,new_symbols;

   old_symbols - symbol table of existing rule

   new_symbols - symbol table of new rule  (with the same name)

   Compare the two symbol tables.  Return TRUE if they are the same, or FALSE
```

otherwise.

### 17.1.3 handle_activities()

```
int handle_activities(activities, rule)
     ACTLIST_PTR activities;
     RULE_PTR rule;
```

activities -- the activity whose runtime objects we want to find.

rule - the rule whose symbol table we will use in extracting the runtime objects.

This routine will create and fill in the structure that represents the activity of the rule passed to it. The routine is passed a linked list of structures, each of which is a template that represents an activity. The routine creates a new linked list in the rule structure passed to it and duplicates the information in the templates of the linked list. Then, for each activity. it links the arguments in the activity structure to the corresponding entry in the symbol table of the rule. In other words, the arguments of each activity will be pointers to the symbol table of the rule. This guarantees that when the runtime objects are bound in the symbol table entries of the rule, the activity of the rule will automatically point to the correct runtime objects.

This routine handles multiple activities and multiple arguments per activity.

### 17.1.4 find_rel()

```
REL_TABLE_PTR find_rel(rel)
     char *rel;
```

rel -- the name of a relation to search for.

This routine searches the relation table to check if a relation whose name is the same as the name passed to it (rel) exists. If it does, a pointer to the relation is returned. Otherwise, a NULL pointer is returned.

### 17.1.5 find_rule_with_params()

```
RULE_PTR find_rule_with_params(name,param_list)
     char *name;
     PARAM_LIST_PTR param_list;

    name -- the name of a rule we want to search for.

    param_list -- a list of paramter types for the rule.

    This routine searches the RuleTable for a rule whose name is the same as
    the name passed to it.  For each rule that  it finds, it checks if all the
    parameters of the rule are of the same types (class) as the parameters in
    the param list passed.  If it finds a rule whose name and parameters
    are the same, it returns a pointer to the rule.
```

### 17.1.6 find_symbol()

```
SYMBOL_PTR find_symbol(symbol,variables)
     char *symbol;
     SYMBOL_PTR variables;

    symbol - the actual symbol in a rule from a pre or
             post condition

    variables - the symbol table for the rule which
                contains all the variables in the rule

    This function searches the symbol table (variables) passed to it to
    determine whether the symbol passed to it is in the table.
    If it is, a pointer to the entry in the symbol table will be returned.
    Otherwise, the NULL pointer will be returned.
```

### 17.1.7 add_queue()

```
void add_queue(que,info)
     QUEUE_PTR que;
     INFO_LEVEL_PTR info;

    que - Either the execution_que or the backward_que.
```

```
    info - This structure holds the information necessary to
           do backward or forward chaining.

    This routine adds the info structure (which contains a pointer to
    a rule) passed to it to the specified que.  It first calls
    look_up_queue() to make sure that the same instances of the rule is
    not already on the queue.
```

### 17.1.8 clear_queue()

```
void clear_queue(queue)
     QUEUE_PTR queue;

   queue -- This is the queue that is cleared out.

   This routine clears out the queue and frees the space
```

### 17.1.9 init_queue()

```
QUEUE_PTR init_queue()

     This routine initializes a queue structure to be used in chaining
     cycle.
```

### 17.1.10 look_up_queue()

```
int look_up_queue(que, info)
     QUEUE_PTR que;
     INFO_LEVEL_PTR info;

    que - which queue should we look into

    info - The structure that contains the rule pointer.  We want to verify
           if an identical structure is already on the queue.

    This function searches a queue to determine if a rule is on that queue.
    The routine first checks if a pointer to the same rule is already on the
```

queue. If it doesn't find one, it looks at all the rules with the same name on the queue. If any of them have the same symbol table and the same activity as the rule passed to the routine, it means that we have two instances of the same rule (this is caused by having multiple arguments that cause chaining to the same rule bu with different runtime objects. The routine distinguishes between different instances of the same rule by calling compare_runtime_objs() to check if the two instances have the same runtime objects. If they do, then they are identical instances and TRUE is returned; otherwise FALSE is returned

### 17.1.11 strsave()

```
char *strsave(string)
char *string;
```

strsave() is used to allocate memory for dynamically allocated strings. It uses the Marvel allocation routine MA_malloc(). If the string passed in is NULL, NULL is returned. If the allocation fails, a messages is printed, and NULL is returned.

A pointer to the new string is returned upon success. Note that this routine is very useful when it is necessary to save the results of a routine that fills in some static buffer, or similar function.

### 17.1.12 add_symbol()

```
int add_symbol(rule, symbol)
     RULE_PTR rule;
     SYMBOL_PTR symbol;
```

rule -- the rule to add the new symbol to

symbol -- the symbol to add

Add a new symbol to the rule's symbol list. If the symbol already exists, and it's type is different then the new symbol, return FALSE, otherwise return TRUE (but don't add duplicate entries).

### 17.1.13 find_symbol_from_binding()

```
SYMBOL_PTR find_symbol_from_binding(rule, binding)
     RULE_PTR rule;
     BINDING_PTR binding;

   rule -- the rule in which to check for the symbol

   binding -- the binding to check against.

   Find a symbol in the rule's symbol list.  Make sure the class is the
   same.  Returns the symbol for success, or NULL.
```

# Chapter 18

# Printing Objectbase and Rule Information

The print module contains code to print answers to simple queries on the Marvel objectbase and process model.

## 18.1 The Print Command

The following functions, all from `print.c`, make up the fundamentals of the calling part of the print command. All but the first are only used in the graphics interface.

### 18.1.1 print_cmd()

```
int print_cmd(cmd_line)
CMD_LINE_PTR cmd_line;

   cmd_line -- structure containing the user command and its arguments

   This routine is the driver of the print module.  It looks at the cmd_line
   structure and extracts the arguments to the print command.  It calls the
   appropriate routine depending on these arguments.  It calls one of
   print_rule(), print_rel() or print_obj() depending on the first argument
   of the print command.
```

### 18.1.2 get_print_graphic_args()

```
int get_print_graphic_args(cmd_line)
CMD_LINE_PTR cmd_line;
```

```
   cmd_line -- structure containing the user command and its arguments
```

### 18.1.3 print_opts_R()

```
int print_opts_R(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

   cmd_line -- structure containing the user command and its arguments
```

### 18.1.4 print_opts_current()

```
/*ARGSUSED*/
int print_opts_current(cmd_line, opt)
CMD_LINE_PTR cmd_line;
int opt;

   cmd_line -- structure containing the user command and its arguments
```

### 18.1.5 print_opts_r()

```
int print_opts_r(cmd_line, opt)
     CMD_LINE_PTR cmd_line;
     int opt;

   cmd_line -- structure containing the user command and its arguments
```

### 18.1.6 print_opts_single()

```
int print_opts_single(cmd_line, opt)
     CMD_LINE_PTR cmd_line;

   cmd_line -- structure containing the user command and its arguments
```

### 18.1.7 print_opts_string()

```
int print_opts_string(prev_arg, opt)
     CMD_LINE_PTR prev_arg;
     int opt;

   cmd_line -- structure containing the user command and its arguments
```

## 18.2 Rule Queries

The following functions contain code to print information about the rules and process model in many different ways. They are all contained in the `print_str.c` file.

### 18.2.1 print_rule()

```
void print_rule(cmd_line, num_args)
     CMD_LINE_PTR cmd_line;
     int num_args;

   Print useful information about rules

   Usage here looks like:
   print -r [<rule> [b[*]]  |          bindings
                    [pr[*]] |          propertylist
                    [a[*]]  |          activitys
                    [po[*]] |          postconditions
                    [c[*]]  |          chains
                    [s[*]]]            strategy

       * just -r prints out all the rules.
       * a <rule> prints out that particular rule and one piece of optional
         information.
```

### 18.2.2 print_bindings()

```
void print_bindings(bindings, level)
     BINDING_PTR bindings;
     int level;
```

```
bindings -- the bindings of a rule.

level -- used to print complex bindings recursively.

This routine prints the bindings of a rule by printing the binding
operator (which can be a universal or existential quantifier, and the
charactersitic function of the binding.
```

### 18.2.3 print_chains()

```
static void print_chains(cond)
     COND_PTR cond;

    cond -- the pre or post condition whose chains will be printed.

    This routine traverses all the predicates of the condition passed to it,
    and for each predicate, it extracts and prints first all the forward
    chains and then all the backward chains.
```

### 18.2.4 print_conditions()

```
void print_conditions(cond, level)
     COND_PTR cond;
     int level;

    cond -- The pre or postcondition that will be printed.

    level -- used to print complex predicates recursively, one level at a
             time.

    This routine prints a predicate condition.  If the condition is complex
    (i.e., it is AND, OR, or NOT of subconditions, the routine calls itself
    recursively on the subconditions after printing the name of the complex
    operator (i.e., AND, OR, or NOT).
```

### 18.2.5 print_entry()

```
void print_entry(pred)
```

```
PRED_TABLE_PTR pred;
```

pred -- The predicate that will be printed.

This routine prints the defintion of a predicate from the predicate table.

### 18.2.6 do_print_activities()

```
static void do_print_activities(rule)
     RULE_PTR rule;
```

rule -- a rule from the rule table

This routine calls print_activity() to print the rule's activity. It handles multiple activities and multiple paramters per activity.

### 18.2.7 do_print_bindings()

```
static void do_print_bindings(rule)
     RULE_PTR rule;
```

rule -- a rule from the rule table

This routine calls print_bindings() to print the bindings of the rule.

### 18.2.8 do_print_chains()

```
static void do_print_chains(rule)
     RULE_PTR rule;
```

rule -- a rule from the rule table.

This routine calls print_chains() to print the forward and backward chains that the predicates in the precondition and postcondition of the rule cause.

### 18.2.9 do_print_posts()

```
static void do_print_posts(rule)
     RULE_PTR rule;

   rule - arule from the rule table.

   This routine prints all the postconditions of a rule by calling
   print_one_post_condition() to print each postcondition.
```

### 18.2.10 do_print_props()

```
static void do_print_props(rule)
     RULE_PTR rule;

    rule -- a rule from the rule table

    This routine calls print_conditions() to print the property list of the
    precondition of the rule.
```

### 18.2.11 do_print_strategies()

```
static void do_print_strategies(rule)
     RULE_PTR rule;

   rule -- a rule from the rule table.

   This routine prints the strategies in which the rule is defined. A rule
   in the rule table can be the result of merging several rules from
   different stratgies which have the same name, paramter types, and
   activities.  In this case, the rule is said to be defined in several
   strategies.
```

### 18.2.12 print_one_post_condition()

```
void print_one_post_condition(cond)
     COND_PTR cond;
```

```
    This routine prints all the predicates of one postcondition.  All the
    predicates of a postcondition are ANDed.
```

### 18.2.13 print_rule_name_and_params()

```
void print_rule_name_and_params(rule)
     RULE_PTR rule;

    rule -- the rule whose name and params will be printed.

    This routine prints the name and parameters of a rule.
```

## 18.3 Relations

This stuff is currently garbage.

### 18.3.1 print_rel()

```
void print_rel(cmd_line, num_args)
     CMD_LINE_PTR cmd_line;
     int num_args;

   Print useful information about relations.

   Usage here looks like:
   print -r [<relation> [s[*] | i[*] ] ]

       * just -r prints out all the rules.
       * a <rule> prints out that particular rule and one piece of optional
         information.
```

## 18.4 Objectbase Queries

The following functions contain code to print information about the objectbase and data model in many different ways. Note when looking at these functions that several print control variables (set with the set command) have a large influence on what is printed out. They are all contained in the `print_ob.c` file.

Printing Information About Classes

### 18.4.1 print_classes()

```
static void print_classes()
```

Print information about all the active classes (those that are not TOOL representations, but rather objectbase representations.

### 18.4.2 print_class_instances()

```
static void print_class_instances(class, space)
     CLASS_PTR class;
     char *space;
```

For the given class, print all the instances. Note that print_kids() will eventually get called here, thus yielding quite a bit more output then might be expected.

### 18.4.3 print_class_subsupers()

```
static void print_class_subsupers(class)
     CLASS_PTR class;
```

Print information about the superclasses and subclasses of the given class.

Printing Information About Objects

### 18.4.4 print_full_objbase()

```
static void print_full_objbase()
```

print the objectbase, recursively.

### 18.4.5 print_obj()

```
int print_obj(cmd_line, num_args)
     CMD_LINE_PTR cmd_line;
     int num_args;
```

Print information about the objectbase.

Here are the arguments:

```
no arguments              -- the current instance.no arguments.
-a                        -- the whole objectbase.
                             no abbreviations to avoid conflicts.
-i                        -- all classes and instances
-c                        -- all classes.
                             Print names of attributes.
                             no abbreviations to avoid conflicts.
<c_name>                  -- Print class, name and value of attributes.
<c_name> <i_name>         -- info about instance listed.
<c_name> <i_name> [<a_name> <i_name>]*
                          -- like above.
<c_name> <i_name> <a_name> [<i_name> <a_name>]*
                          -- small attribute, prints the value.
                             medium attribute, prints path.
                             large attribute, print all elements in
                             set.
```

Three external marvel variables are used, accessable via the set command:

```
    set allmatches -- get the maximum amount of information.
    set depth <number> -- the depth of recursion when printing hierarchy.
    set lines -- the number of lines to print before stopping and
                 requesting more.
```

The code here is very straightforward, and provides a good example for how to traverse almost all of the paths in the objectbase. Note that most of these routines are static.

### 18.4.6 print_kids()

```
static void print_kids(inst, _depth, space)
     INSTANCE_PTR inst;
     int _depth;
```

```
    char *space;
```

Print information about the children, recursively of the given instance. _depth is a controller that determines how far down to recurse, based upon the print controller depth. This routine is the central facility for printing about all the objects in a tree in a recursive fashion.

### 18.4.7 print_instances()

```
static void print_instances()
```

Print information about each instance. Just print the class name, and all the instances. Don't print a class name if there are no instances.

### 18.4.8 print_current()

```
void print_current()
```

Print the current instance, and all of it's hierarchical parents.

## Printing Information About Attributes

### 18.4.9 print_class_attributes()

```
static void print_class_attributes(class, space)
    CLASS_PTR class;
    char *space;
```

Print all the information about the small, medium and then large attributes of the given class.

### 18.4.10 print_instance_attributes()

```
static void print_instance_attributes(inst, space)
INSTANCE_PTR inst;
```

```
char *space;
```

Print all the information about the small, medium and then large attributes of the given instance.

### 18.4.11 print_small_medium_attribute()

```
static void print_small_medium_attribute(att, space)
ATTRIBUTE_PTR att;
char *space;
```

Print information about the given small or medium attribute: its name, type and default value. Use this routine to print information about attributes of classes and objects (instances).

### 18.4.12 print_large_attribute()

```
static void print_large_attribute(att, space)
     ATTRIBUTE_PTR att;
     char *space;
```

Print the name, set or seq information, type and default value of this large attribute. Use this routine to print information about attributes of both classes and objects (instances).

### 18.4.13 print_att_defaultval()

```
static void print_att_defaultval(att, space)
ATTRIBUTE_PTR att;
char *space;
```

Print the default value of an attribute.

### 18.4.14 print_attype()

```
static void print_attype(att)
ATTRIBUTE_PTR att;
```

Print the type of attribute this is.

### 18.4.15 print_set_seq()

```
static void print_set_seq(att)
ATTRIBUTE_PTR att;
```

Print whether the given attribute is a set or sequence.

### 18.4.16 get_type_name()

```
static char
  *get_type_name(typ)
int typ;
```

This function, given a type code, returns a string representation of the type's name.

## Printing Information About Links

### 18.4.17 print_link()

```
static void print_link(link, space)
LINK_PTR link;
char *space;
```

Print information about the given forward link.

### 18.4.18 print_links()

```
static void print_links(att, space)
ATTRIBUTE_PTR att;
```

```
char *space;
```

Print all the links of this attribute.

### 18.4.19 print_blinks()

```
static void print_blinks(own_link, space)
OWN_LINK_PTR own_link;
char *space;
```

This function prints out the links which point to a given instance or attribute. It does not check the global flag "printblink" before doing this; that is the responsibility of the calling procedures. So these are the back links.

Printing from Mouse Info Picks

### 18.4.20 immead_print_instance()

```
void immead_print_instance(inst)
INSTANCE_PTR inst;
```

Print all the information about a particular instance, just as it would have been printed in a larger list. This is used by the browser, when a user clicks upon an object to get information about that object.

Printing Single Entities

### 18.4.21 print_particular_class()

```
static void print_particular_class(c_name)
char *c_name;
```

Print information about the particular class described by c_name.

### 18.4.22 print_particular_inst_of_attribute_with_path()

```
static void print_particular_inst_of_attribute_with_path(cmd_line, num_args)
     CMD_LINE_PTR cmd_line;
     int num_args;
```

### 18.4.23 print_particular_inst_of_class()

```
static void print_particular_inst_of_class(cmd_line, num_args)
     CMD_LINE_PTR cmd_line;
     int num_args;
```

### 18.4.24 print_particular_inst_of_class_with_path()

```
static void print_particular_inst_of_class_with_path(cmd_line, num_args)
     CMD_LINE_PTR cmd_line;
     int num_args;
```

Line Control for Command Line Interface

### 18.4.25 line_incr()

```
static void line_incr()
```

Also incr_batch_return() macro.

Increment a static counter that determines if it is time to stop printing output in the line oriented user interface. This routine's action is affected by all the various print controllers for the line oriented interface. It is never called in the X11 interface, because of the macro incr_batch_return above.

This routine sets the time_to_go flag is set to TRUE if it is time to stop printing. This flag is automatically checked with the macro below.
It should also be checked after returns from function calls, in which it

might have been set.

The incr_batch_return macro checks for the line interface, and if so, does a line_incr, then returns if it is time to do so.

# Bibliography


[BK88] Naser S. Barghouti and Gail E. Kaiser. Implementation of a knowledge-based programming environment. In *21st Annual Hawaii International Conference on System Sciences*, volume II, pages 54–63, Kona HI, January 1988. IEEE Computer Society.

[CSB89] Mara W. Cohen, Michael H. Sokolsky, and Naser S. Barghouti. Marvel 2.5 user manual. Technical Report CUCS-498-89, Columbia University Department of Computer Science, December 1989.

[FK87] Peter H. Feiler and Gail E. Kaiser. Granularity issues in a knowledge-based programming environment. *Information and Software Technology*, 29(10):531–539, December 1987.

[GEKP88] Peter H. Feiler Gail E. Kaiser and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[GEKS88] Peter H. Feiler Gail E. Kaiser, Naser S. Barghouti and Robert W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[GEKS90] Naser S. Barghouti Gail E. Kaiser and Michael H. Sokolsky. Preliminary experience with process modeling in the marvel software development environment kernel. In *23rd Annual Hawaii International Conference on System Sciences*, Kona HI, January 1990. To appear.

[KF87] Gail E. Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *9th International Conference on Software Engineering*, pages 180–188, Monterey CA, March 1987. IEEE Computer Society.

[SK89] Michael H. Sokolsky and Gail E. Kaiser. Data migration in software development environments. Technical Report CUCS-447-89, Columbia University Department of Computer Science, July 1989.

[Sok89] Michael H. Sokolsky. Data migration in an object-oriented software development environment. Master's thesis, Columbia University Department of Computer Science, April 1989, Technical Report CUCS-424-89.