Marvel 2.5
User Manual

Mara W. Cohen
Michael H. Sokolsky
Naser S. Barghouti

Columbia University

TR CUCS-498-89

March 14, 1990

Programming Systems research is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research. Sokolsky is supported in part by C.A.T. Barghouti is supported in part by C.T.R.

Contents

List of Figures

1 Introduction

This manual is about MARVEL, a knowledge-based software development environment. Several papers present the idea and the architecture of MARVEL [4, 2, 1, 3]; this manual concentrates on teaching the user how to use the system. The manual consists of several parts, including: a tutorial which walks the user through an example, a reference section that documents each built-in feature of MARVEL, and an appendix that provides a guide to the source code, and copies of the strategies and envelopes used as examples in this manual.

1.1 Who Would Use This Manual

There are basically two types of people for whom this manual might be useful. The first are students who want to learn the C programming language and know nothing about UNIX. This manual will provide these students with an environment in which to learn C without having to also spend time learning about UNIX. The second group of people who might find this manual useful are those who are interested in MARVEL for research purposes.

1.2 Typographical Conventions

In order to make this manual more readable there are certain typographical conventions that you will notice and should understand. First, you probably have already noticed that every time the word MARVEL appears it has a special font; this is because it is the name of the program that you are using. Any commands, rules, or prompts that MARVEL shows you are in

boldface font and set off from the rest of the print,

so that you can easily identify them. Anything that you will be typing in response to a prompt is in

italics and set-off from the rest of the print

so that it is different from the MARVEL prompts and identifiable. The programs that you write are in a

```
typewritten type of script
```

so that you will notice them as well. Anything between angled brackets, <>, is the generic name for that thing which is different from the specific name. For example, any object would be shown as < *objectname* >.

1.3 About MARVEL

The idea of MARVEL was conceived at the Software Engineering Institute of Carnegie Mellon University by Peter H. Feiler and Gail E. Kaiser in the summer of 1986. A "proof-of-concept" implementation was done at the SEI at that time by Popovich. The first large-scale implementation of MARVEL started at Columbia University in the fall of 1986. During the summer of 1987, the project was a joint one with Siemens Research and Technology Laboratory. This implementation, as well as the SEI implementation, was built on top of a multiuser programming environment for C called Smile[5] and was completed in September 1987. The second large-scale implementation, which this manual presents, began in September 1987 and was completed in February 1990.

The current implementation is completely standalone with a persistent object manager replacing Smile. In no way should this object manager be confused with a generalized objectbase facility, as we recognize the lack of key concepts such as concurrency control. But it is appropriate to explore the automatic environment generation facilities of MARVEL in a real, usable system. MARVEL consists of two executable programs, the kernel and a separate strategy loader. The strategy loader operates in a separate process to facilitate basic fault tolerance when changing environments and debugging strategies, for only a very small performance penalty which the authors feel is acceptable for the current experimental state of the system.

The basic idea of MARVEL is to automatically generate an environment to support the specific needs of people working on a software project. MARVEL provides facilities to generate such an environment in two phases. First, a person called the MARVEL *administrator* writes a description of a software project. The description

- specifies the organization of the objectbase containing the components of the project in terms of classes and attributes. For example, a C program may consist of modules (each of which may contain macros, types, variables and functions), documentation, and test suites. In the literature referenced above, this is technically called the data model;

- models the software development process for that particular project in terms of rules with preconditions and postconditions. For example, an editor (rule) can have a precondition that the specified module be assigned to the current programmer, and a postcondition (result of the editor) be that the module's status is not-checked, implying it is necessary to invoke a type checker. These preconditions and postconditions form rules for the operation of the environment, this is technically the process model;

- provides mappings in the rules to envelopes, which in turn can call arbitrary COTS (commercial off the shelf) tools or local tools.

Second, a person called the *user* starts up MARVEL and loads an appropriate subset of the description to create MARVEL's kernel. The kernel incorporates the description, tailoring its behavior according to the model of the software process depicted in the description. The kernel includes an objectbase manager that understands the organization part of the description, and tailors its model of the data according to it. The result is a tailored MARVEL that the user can use to develop the target software system which is called the MARVEL environment.

Figure 1 depicts the generation of a MARVEL environment. MARVEL's kernel supplies the *meta-knowledge* shared by all MARVEL environments, in particular how to construct and maintain the objectbase and how to do controlled automation. In the following sections, we guide you through an example in which you generate your own programming environment that behaves according to the descriptions you provide MARVEL.

MARVEL has two types of interfaces: a command line interface, and one using X11-windows (version 3 or 4). Chapter 4 explains how to use MARVEL with each of these interfaces. The tutorial presumes that the reader is using the X-windows interface. However, both interfaces have equivalent power, thus the same objects could be created in the line interface.

1.4 About This Manual

Much of the information in this manual is presented in the format of a tutorial of MARVEL. The tutorial is included with the distribution system. Then there are a number of sections on advanced topics, and a reference section that contains manual pages for all the built in marvel commands. Finally, there are appendices that document all the strategies used in the tutorial, and provide a software map for MARVEL.

Please note that MARVEL is provided essentially free and "as is". This means that the authors are not responsible for providing any particular support or enhancements. At this point, this is just not feasible with only two full time people working on the system. We appreciate any feedback, but caution users not to expect fast response, if any response.

1.5 Some Useful Terminology

This manual will be using common computer science terminology with which you may not be acquainted but which you will find useful. The development of the objectbase is akin to the growth of a tree. The object that is highest in the class structure is called the **root**. In Figure 2 the root is the object labeled **top**. The root is represented at the top of the tree and the rest of the tree grows down from it. In this manual,

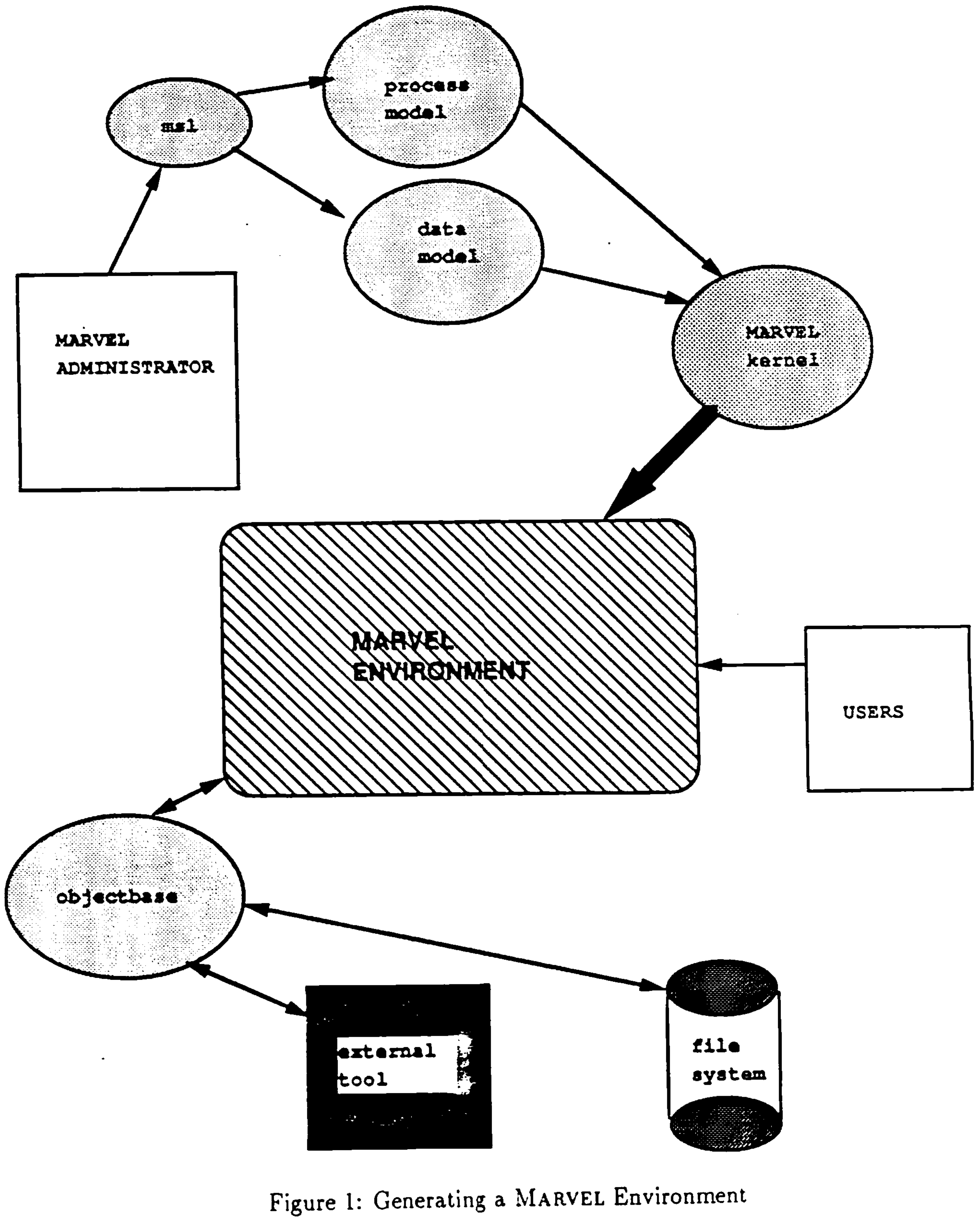

Figure 1: Generating a MARVEL Environment

objects can be composite objects; any object that has other objects descending from it is called a **parent** and those descendants are called **children**. Again observe Figure 2, the root **top** is a parent of both **proj** and **proj2**. **Proj** is in turn a parent to **prog**. Looking at this another way: **prog** is a child of **proj** and **proj** is a child of **top**. When you look at the representation of your objectbase as it grows you will notice that it does resemble an upside-down tree. A MARVEL objectbase can contain any number of trees, thus making a **forest**, just as many trees do in nature. However, the entire objectbase can be viewed as a directed graph, as well, because directed links between arbitrary objects are supported.

1.6 Preparing to Run MARVEL

This section includes a few things that need to be known in order to successfully run MARVEL.

1.6.1 Unloading the Distribution Tape

Skip this section if MARVEL has already been installed on your system.

1. Select a user id. If you unload the tape as root, all the files will probably be owned by a user not on your system. Otherwise, the user who unloads the tape will own all the files;
2. Find an appropriate tape drive. The Sun tapes were made on a standard Sun drive on a sun4. The IBM tapes were made on a IBM 6157-002. The DEC tapes were made on a TK50;
3. cd to a directory under which you desire to have the MARVEL release reside. There must not exist a directory or file called `marvel` there;
4. Insert the distribution tape into the tape drive;
5. Unload the tape with tar:

   ```
   tar xf <raw device of tape>
   ```
6. Remove the tape;

1.6.2 MARVEL Environment Variables

There are several environment variables which need to be set before attempting to run MARVEL:

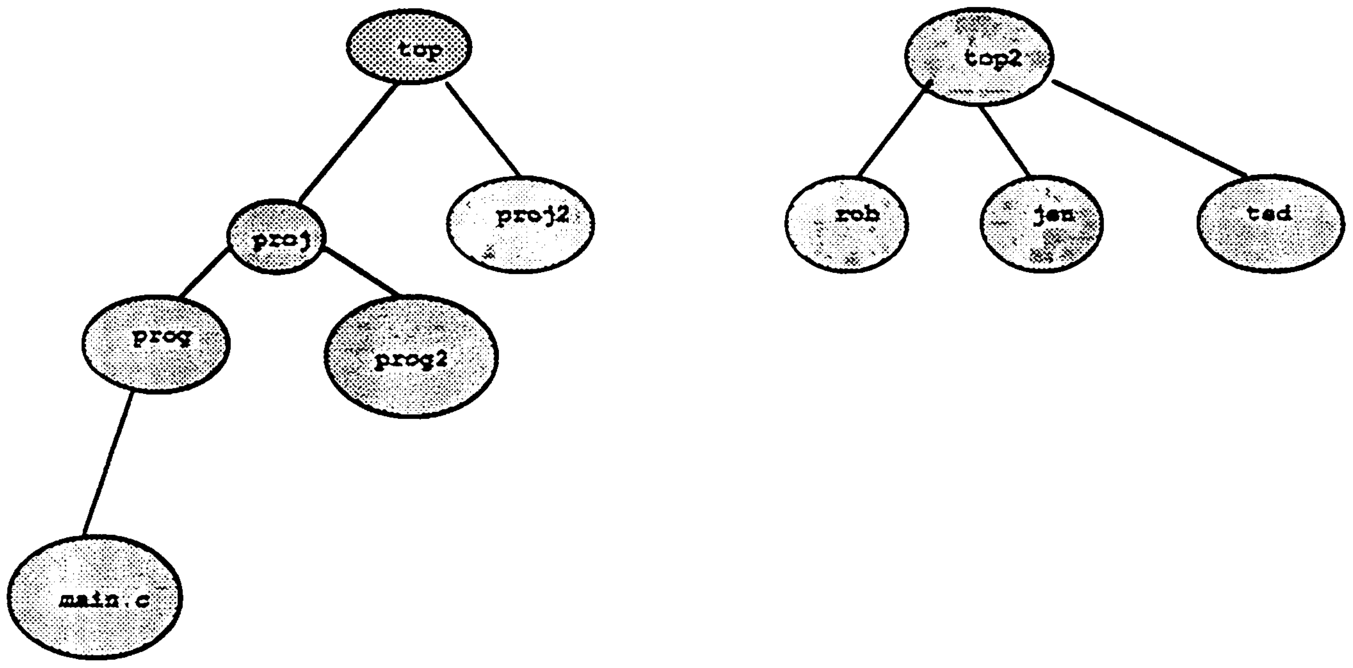

Figure 2: A MARVEL Objectbase

MARVEL_HELP_DIR A directory where MARVEL looks for manual pages. At this point, this can not be a list of directories. The default is

/proj/marvel/help

MARVEL_DB_ROOT The name of the "top level" class in the data model. The default is the for the *C/Marvel* environment, namely **GROUP**.

MARVEL_LOADER The path to the loader program. The default is

/proj/marvel/bin/loader

MARVEL_HELP_EDITOR A path to a text filter for displaying manual pages. The default is

/usr/ucb/more

MARVEL will complain and not start up if it can't find any of the information described above. It is most convenient if these all be set up in each user's home directory.

1.6.3 The `.marvelrc` File

Each user can have a file called .marvelrc in their home directory. This file should contain MARVEL commands, and they will be executed upon startup. This startup file is a good location to put set commands for setting MARVEL variables, and other common initialization tasks.

2 A Tutorial

This section is an interactive tutorial of MARVEL . Many of the features of the system are shown here through a simple example. This step by step tutorial will take you through the creation of a simple C program. It is most helpful to run MARVEL as suggested before attempting to use it on your own work. The tutorial should not take long to follow through. This tutorial presumes that you will be using the X-windows interface; there exists a command line interface that is explained in Chapter 4.

2.1 Tools

The following example uses MARVEL to help with the development process using the C programming language for developing a system. In order for MARVEL to help you in the development process, it has to understand how you want to structure your project and how the tools behave. Knowing the structure enables MARVEL to organize the components of the project and do all the book-keeping that is necessary. The behavior of the tools and how they affect the various components provides MARVEL with the knowledge of when to use these tools and when they cannot be used. The tutorial will use *Cmarvel*, a MARVEL environment that comes with the distribution tape.

2.1.1 How to Tell MARVEL About Structure and Behavior

Having decided on the structure of the project (data model) and the tools (process model) you will use in developing it, you now need to tell MARVEL about them. Unfortunately, MARVEL does not read the user's mind nor does it understand natural language. To solve this problem, we have developed an object-oriented language called MARVEL Strategy Language (MSL), in which descriptions are written. The syntax and semantics of MSL are described in Chapter 12 of this manual.

The first step in using MARVEL is to write a description of the project environment. The details of this can be found in the MSL manual. This is beyond the scope of this manual, we will use a MARVEL environment called *C/Marvel* that has been developed to test MARVEL and comes with the release. We briefly explain the description of the example project with which this tutorial deals. The description is modularized into units called *strategies* where each strategy gives a subset of the complete description. Several strategies can be merged to produce the desired structure and behavior.

The MSL definitions that comprise *C/Marvel* are provided in Appendix A. The strategy `cmarvel_chaining` is a "root" strategy, in that all it does is "import" enough other strategies to create an environment. The strategy `data_model` describes the data model of *Cmarvel*, and the remainder of the strategies describe the rules and tools available with *C/Marvel*.

The classes are connected together hierarchically through the attributes that are of type *set_of* (called *set attributes* hereafter). For example, the PROJECT class has an attribute called *modules* which is defined to be of type *set_of MODULE.* This means that each object of type PROJECT will contain a set of modules. Similarly, each object of type MODULE contains a set of procedures. (For more information on this see Chapter 2.2.) There are several links of varying types in order to establish non-hierarchical connections.

Describing tools and their behavior is slightly more complicated since we need to first describe the tool in terms of the operations that it can perform (some tools can be used to perform several operations). These operations are called *activities.* We also need to describe the behavior of the tool in terms of the *precondition* of invoking it and the several alternative results (called *postconditions* hereafter) of invoking it. Finally, we need to specify how to obtain the input of the tool from the repository which contains the components of the project, and how to map the outputs of the tool into the repository. This is done with tool *envelopes*, which are in Appendix B.

We mentioned that strategies can import other strategies. This means that the basic structure of the environment can be defined in one strategy, and then several other strategies which define various kinds of behavior can all share this structure. Such importing can (and should) be recursive. Marvel does not currently support circular definitions of imports.

We now describe how to set up the project and start using the *C/Marvel* environment to obtain the assistance that you need. Later on in the manual we describe how to alter the behavior of MARVEL dynamically by switching strategies.

2.1.2 Behavior of the Tools Used to Develop the Example Project

The easiest way to describe the behavior of a tool is to specify when the tool can be invoked (i.e. the condition that must hold for the tool to be applicable), what the tool expects as inputs, and what changes the tool affects on the components of the project, if any. Let us assume that the tools that you will be using in this example behave as follows:

arch_proj This tools archives the project after making sure that all the libraries are archived.

arch_mod This tool runs the archiver if all the C files are analyzed.

arch_lib This tool archives all the libraries if all the modules are archived.

analyze This tool is used to analyze the code of a procedure. It is invoked only if the procedure has been edited since the last time it was analyzed. In the case of a successful analysis, the procedure is marked as *Analyzed*; otherwise, it is marked as *NotAnalyzed* to indicate that it needs to be modified (by using the editor). Furthermore, the module in which the procedure exists is marked as *NotCompiled* in the case of a successful analysis.

build_all This tool calls the routine that builds all the programs in a project.

build This tool links all the components of the program and produces the executable. It can be invoked only if all the cfiles of the program have been compiled. If the building is successful, the project is marked *Built*; otherwise, it is marked as *NotBuilt*.

compile This tool is used to compile a module. It can be invoked only if all the procedures in the module have been analyzed and if at least one of these procedures has been modified since the last successful compilation of the module. If the compilation is successful, the module is marked as *ModIsComp*.

deposit This tool is used to lock files. A file is deposited so that a back-up copy of the file is created and, therefore, if a file is lost this back-up will remain and a copy will still exist.

debug This tool is used to debug and execute PROGRAM type objects. It requires the PROGRAM to have the built_status equal to BUILT or the debug_status equal to NeedsDebugging in order to fire..

exec_prog This tools executes the executable files.

edit This tool is used to modify the code of a procedure. It can only be invoked when the user desires; it expects the name of the object to be edited. It will not work unless the file to be edited is reserved by the user. Edit is an *overloaded* rule which differentiates between various usages of the same symbol.

list_arch This tool lists the contents of an archive file.

reserve This tool is used to unlock files that have been deposited. A file is reserved by you so that only you can alter. When you are finished with this file you deposit it and other people can then use and alter it.

release Currently, this tools is non-functional.

ViewCerr This tools displays all the compilation errors.

ViewAerr This tools displays the analysis of the errors.

view This tools displays the text of the object.

2.2 The Structure of MARVEL

This section describes the class hierarchy of MARVEL using the data model of the *C/Marvel* environment. This hierarchy is very important because, as you shall see, all the work that you do depends upon how the objects you create relate to each other and how the attributes that relate to these objects allow you to do certain things. Figure 3 will help you follow the discussion in this section.

The highest class on the hierarchy is called **GROUP**; this is the "root" of the "tree". Since you can create a forest, there can be more than one object of the class GROUP. GROUP has children which are classified as **projects**. These **projects** in the class PROJECT are actually called a set_of PROJECT (which is how all the classes are defined) and describes the structure of a software project. It contains the following subsets: **libraries**, **bin**, **programs**, **doc**, and **incs**. The **libraries**, class LIB, is a share archive type library and consists of **modules**. The class PROGRAM is different from PROJECT even though it does parent some of the same objects. The difference is that PROGRAM is a single executable unit while PROJECT is a collection of entities. The children of PROGRAM are: **docs**, **cfile**, **modules**, **incs**, and **ofiles**. A MODULE is basically just a unit for organizing **cfile** and **hfiles**. A CFILE is used to contain a C program and allows for the recording of pertinent information like the state of compilation and analysis. The HFILEs are the include files in a C program. The class DOC represents a set of DOCUMENTs and they are exactly what they sound like, files to text which is documentation for the project on which you are working. Sets of HFILEs are represented by the INC class. Finally, BIN represents the place where binaries for PROGRAMs are kept.

You have seen two diagrams that look very similar, Figure 2 and Figure 3. The first one is an objectbase which is your model of your project. The second, the data model, show you have the various classes fit together in the hierarchical structure. Of course, there will be much similarity between your objectbase and the data model since the objectbase is based upon the structure of the data model. Therefore, the data model is the template for the design of the objectbase.

2.3 Creating a MARVEL Objectbase

As we have stated before, this tutorial is written presuming that you are using the X-windows interface. How X-windows works is not discussed in this manual; we suggest that you feel comfortable with the X-windows before you begin using MARVEL. MARVEL assumes you have the window system started, and that you are working in a text window.

Before you even begin MARVEL you must create a place for yourself to work. This is extremely simple. First go to some appropriate place, like a directory that you have created to hold all of your MARVEL work. Then type

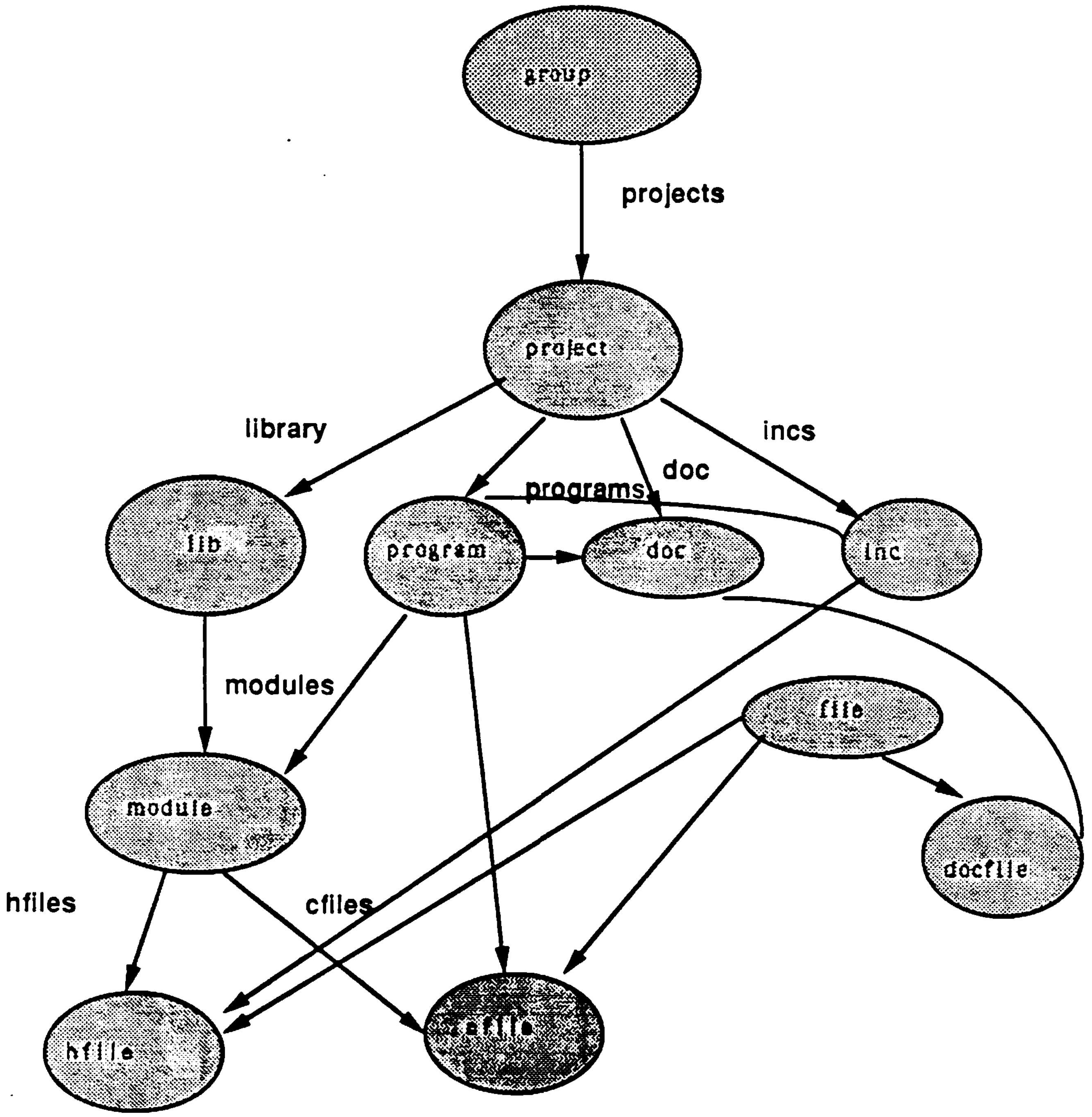

Figure 3: The Data Model *C/Marvel*

```
make_db new.
```

Make_db is a script that creates the directory in which you will be working and the directory in which your data will be stored. **New** is the directory in which the our example will be run and that we call a MARVEL database. After make_db has finished it reminds you to copy in your strategies and envelopes. Now type

```
get_cmarvel new
```

which is a script that copies in the appropriate strategies and envelopes.

Start MARVEL by typing

```
marvel -w.
```

If you have an objectbase but are not in its directory then you will be queried with

```
objectbase name (q to exit)?
```

To which you respond with the name of the appropriate objectbase. In our case this is new.

If you start MARVEL when you are in an objectbase then you will not be asked which objectbase you wish to use; MARVEL presumes that you will use that objectbase. However, if you are not in an objectbase then you will be queried.

2.4 The Windows

There are two windows that you will be using. The one from which MARVEL is called is the **start-up window**. In this window many of the status checks are written. The importance of these checks and how you interact with them is explained throughout the tutorial. From this window the second window, the MARVEL window, pops up. This section explains about the MARVEL window. You will use the mouse to move around the MARVEL window. The chapter on user interfaces describes in detail all of the uses that the mouse has; here let us which both imply that you place the mouse pointer on the particular object or command and press a button.

The MARVEL window should appear (see Fig. 4). As you can see, the window is divided into five parts. The status window (see Fig. 5) tells you the current version of MARVEL that you are using, the current command that you are using, and the current object. The section labeled **display window** contains the diagram of the relationships between the nodes of the forest that you are going to build. The section labeled **text window** will prompt you with questions and keep you notified about the status

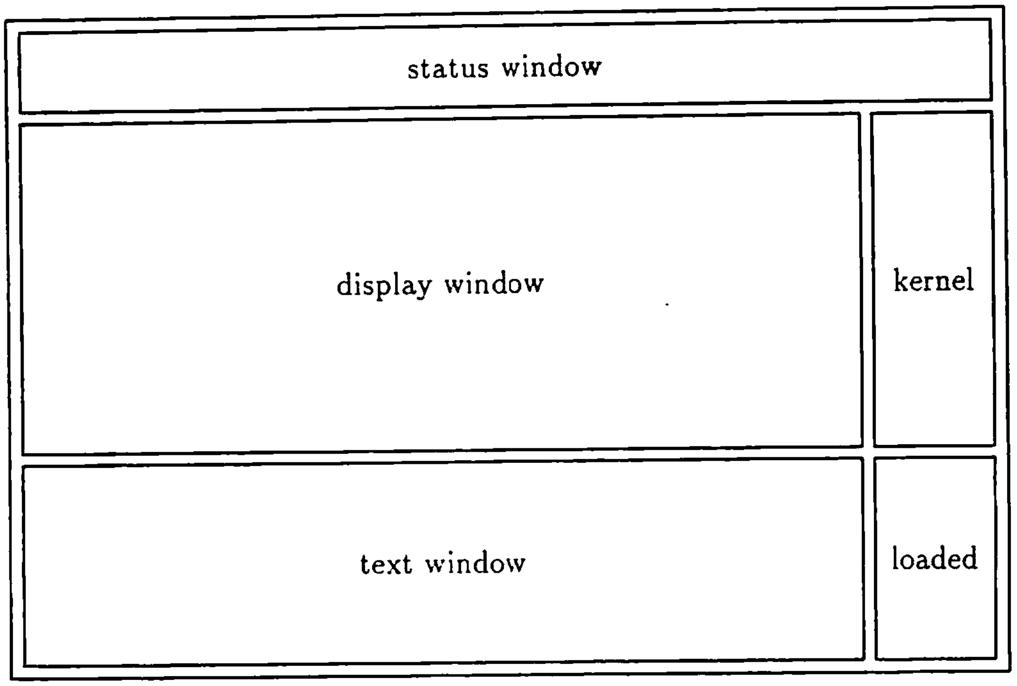

Figure 4: Layout of the Marvel Screen

of what you are doing. The **kernel commands** list the built-in commands (which are explained in Chapters 6, 7, and 2.7) and the section labeled **loaded commands** contains strategies for C/MARVEL. If you are starting out without ever having done anything on MARVEL before then there should be no **rules**. This is because you have not loaded a subset of a MARVEL environment.

To load a set of strategies from C/MARVEL choose the **load** box. The text window will prompt you

Enter strategy name:

Type in

cmarvel_chaining

When **load** is finished the **loaded commands** should be offering you a choice of options like those reviewed in Chapter 8.1 and you can begin designing your objectbase.

Marvel 2.5	current command	current object

Figure 5: The Status Window

2.5 Designing the Objectbase

Now you want to begin your sample C program. This section will take you through a step-by-step explanation of building a small objectbase. To start your tree follow these steps:

1. click on **add**
   (You can use any of the buttons on the mouse during this entire process.)

2. choose the option **horiz**

3. look at the **text window** and to

   Enter <instance>:

   reply

   top

   (the pointer can be anywhere in the MARVEL window for you to write in the **text window**)

4. You will be prompted with:

   Add instance top? [pick yes or no]

   and you pick the **yes** option on the menu. (You have just created the root of your tree and you will now begin creating the rest of the tree.)

5. again choose the **add** option

6. this time choose the option **hier**

7. to the prompt

   Enter <attribute><instance>:

   you reply

   projects proj

8. On the text window you will be asked

   Add instance proj to class PROJECT? [pick yes or no]

   and you choose the **yes** option on the menu. (Notice that not only are the two objects present on the **display window** but a line connecting the two is also there. This line represents the hierarchical relationship.)

9. In order to continue adding objects you must move down this hierarchy, so click on the **change** box

10. On the **text window** you will be asked

    Pick an object to change to.

    and you move your pointer to click on the object **proj**.

11. once again choose **add**

12. and again **hier**

13. this time reply

    programs prog

    and respond **yes**. You have added at the bottom of your tree an object of the class PROGRAMS

14. do not forget to move down the hierarchy as we did before in steps 9 and 10, otherwise you will not be able to add the next object; however, this time click on **prog** since you want to add a level below it

15. once more choose **add**

16. again pick **hier**

17. this time reply

    cfiles main.c

    and respond **yes**

(for more information on **add** see Section 7.1) The name of this last object should look somewhat familiar. You have just created the object in which you will write your C program.

2.6 Editing an Object

Before the object can be edited you need to use the **reserve** tool. This will prevent anyone else from tampering with the file while you are editing it. To reserve the file **main.c** you must use the **start-up window**. The best way to do this is to situate this window and the MARVEL window so that you can see both at the same time. In the **kernel commands** choose the option to **edit**. The **text window** will query with:

Enter rule arguments:

and you respond

main.c

or you can pick **main.c** by clicking on it with the mouse. You will then see in the **text window** that the precondition to edit is not satisfied and some chaining will occur (see Sections 2.7 and 2.8 for more information about chaining). Now an edit window should appear with your file waiting to be edited. The editor that you use will depend upon to what your EDITOR shell variable is set. If this is not defined then your editor will be `vi`. This manual does not explain anything about how to use any type of editor, it presumes that you have a favorite one that you use and with which you feel comfortable.

The editor that is specified in the editor envelope will be invoked. At this point, you can add the code of the procedure or modify it if already contains code (if this not the first time that you edit this procedure). Since we are more concerned with learning about MARVEL than with programming write a simple C program, such as:

```
main()
{printf(''hello world\n'');}
```

Now save this program. When you exit the editor MARVEL automatically prepares you for the next step.

Writing a program without seeing if it actually works seems somewhat silly, so now compile your program. To do this click on the **compile** box in the list of **loaded commands**. Choose the option of **compile**. The **text window** will prompt you with:

Enter rule arguments:

and type

main.c

You can watch as MARVEL does it's chaining. First it checks to make sure that the precondition for the analyze rule is satisfied, then it shows the options it finds for forward chaining and whether or not they are satisfied. MARVEL should choose the compile option and forward chain to that and then continue on to the build rule. Finally, you will see:

Execution cycle completed.

In the **start-up window** the results should appear:

hello world

The next section explains the concept of chaining and how it relates to MARVEL.

2.7 Forward chaining

Forward chaining is based on the preconditions and postconditions of the rules that run the program. The idea is that when a postcondition for a rule is satisfied the precondition for the following rule has been satisfied. In other words, the postcondition of the first rule is equal to the precondition of the second rule, and so on. This postcondition-precondition equality continues ad infinitum until there are no rules left to fire or a precondition is not satisfied.

When you quit the editor, the postcondition of the edit rule is asserted (the following should appear in the **text window**):

Execution cycle completed.

This assertion satisfies the precondition of the analyze rule and the analyze rule fires automatically. Thus, the analyze envelope will be executed and lint will be invoked on the procedure that you have just edited. What happens afterwards depends on the result of the analysis. If lint returns no errors, the *analyzed* attribute of the procedure will be set to "Analyzed" and the *edited* attribute will be set to "NotEdited". Changing the value of the *analyzed* attribute will cause the precondition of the compile rule to be evaluated.

The precondition of the compile rule states that the activity will be invoked if all the procedures that are members of the *procs* attribute of a module are analyzed. Thus, when a procedure is analyzed, the precondition of the compile rule is evaluated after binding the module variable (?m in the compile rule) to the module that contains the procedure that was analyzed. If all the other procedures of this module have been analyzed, then the module will be automatically compiled.

2.8 Backward chaining

Forward chaining is applied to automatically invoke tools after their preconditions become satisfied. Alternatively, backward chaining is used to invoke tools in attempting to satisfy the precondition of a rule that the user invoked. MARVEL will inform you if the precondition of the compile rule is not satisfied usually because the procedure is not analyzed. It then tries to execute the analyze rule to satisfy that precondition, but finds out that the procedure cannot be analyzed because it has not been edited. At this point, MARVEL will terminate its backward chain because it cannot invoke the editor automatically (because the editor has no precondition which indicates that it can only be invoked by the user). Note that you could modify the edit rule to contain such a precondition if desired. Then edit would automatically be invoked upon detection of compiler or analysis errors.

We call cycles of chains **execution cycles**. A cycle can consist of either backward or forward chains, not both.

3 Advanced Tutorial

This tutorial is for people who wish to learn a bit more about how the class structure works. We presume that you have completed and understand the previous example because we will build upon that knowledge and that example without reviewing it

4 User Interfaces

The command line interface for MARVEL does not use X-windows. It is based on the user responding to a prompt and MARVEL responding to that command. The difficulty with this interface is that it does not pictorially describe the hierarchy or graph structure of the objectbase. The X-windows interface allows for graphics. With a graphics interface a picture of the objectbase can be seen which is very helpful in understanding exactly what you are doing and how objects relate to each other. The graphics interface also allows for a MARVEL screen which helps to take the onus off you, the user, because the commands and the rules for each command are present on the screen and you do not have to remember all the commands and which rules apply to which commands.

4.1 Command Line Interface

All of the commands that are discussed in this section are also available with the graphics interface. The usages of these commands can be found in Chapter 13.

print This command prints information about the objectbase and the data process.

add This command adds an object to the objectbase. Given only a name, it will add an object of the current class with the given name. Otherwise, it can add an object hierarchically to a set attribute of the current object.

link This command is used for referring to things that are elsewhere in the objectbase but not in the hierarchy.

unlink This command undoes the linking that is done by the **link** command.

change This command is used to browse the objectbase by changing from one instance to another. This change can either be within the same class or across classes, as with **add**.

execute This command is used to execute a MARVEL command script.

load This command uses a list of strategy names as an argument. It will first unload all the currently loaded strategies and then load the listed strategies, merging them recursively. MARVEL will automatically load the strategies that are imported by the specified strategy. This command could change the behavior and structure of the environment depending on the new strategies that are loaded.

merge This command is used to merge new strategies with the currently loaded strategies. Unlike the case with the load command, the current strategies are not unloaded.

unload This command will unload all the strategies listed in the argument. It will also prompt the user if they want to unload all the strategies that use the strategies listed.

copy This command is used by the superuser to copy an object, it's children, and it's attributes from one place to some place else.

move This command moves an object from one parent object to another.

join This command is used to reshape the hierarchy without using copy or move.

rename This command is used to rename an object.

delete This command is used to delete an object.

marvelize This command is used to migrate existing software systems into MARVEL.

set This command is used to set MARVEL environment variables that control browsing in the objectbase. These variables can be preset by the user in their *.marvel_rc* file which MARVEL automatically executes upon start-up.

prompt This command is used to change the MARVEL prompt.

save This command is used to update the objectbase by saving all the new additions made to it since the last save.

readob This command is used to read an objectbase into memory. It is primarily a debugging command for experienced MARVEL users.

? This command is used to display the list of available commands. This list includes all the built-in commands and all the commands that are currently loaded entered as rules.

help This command is used to display help about the command or subject that the user requested. Otherwise, it will display general help about MARVEL. If a "?" is given, a list of commands and subjects about which help can be found is given.

usage This command is used to find the usage of a MARVEL built-in command.

quit This command is used to leave the system.

4.2 Graphics Interface

With the mouse and the menus using MARVEL is fairly simple. By placing the pointer on the option (either a **kernel command** or a **loaded command**) that you wish and pressing on any of mouse's buttons, the option darkens and what ever you have chosen should happen. We call this state the **idle state** because you are not within a command. When you within a command we call that the **command state**. There are shorter ways to do some of the options on the menu by using the buttons on the mouse.

You will notice that MARVEL provides much information in the **text window**, often too fast for you to follow. Do not be concerned. Immediately next to the **loaded commands** in the **text window** is a **scroll bar** (the narrow column with the arrows at the top and the bottom). By holding your finger on the button of the mouse, placing the arrow on the white box in the column, and dragging the arrow up and down the column the **text window** will scroll. Now you can read all the information at your leisure. To locate a **loaded command** that does not seem to be listed use the **Up** and **Down** box on the menu list.

4.2.1 Left Button In Idle State

When you press the left button the attributes of the object to which you are nearest will scroll on the **text window**. You can be anywhere in the area of the object to get this information. This is the shortcut for the **browse** command with the **info** option.

4.2.2 Right Button In Idle State

By using the right button you can accomplish the same thing as the **change** option on the menu, the current object will be changed. Again, so long as you are in the area of the object the current object is altered.

4.2.3 In Command States

command state: browse In stead of choosing the **pan** option and then selecting **right** or **left**, once you are in **browse** you can put the pointer to the right or left of the object from which you want to pan, press the button and you will receive the information you requested.

command state: change When you select the **change** option click on the object that you want to change to and the **Current Object** will change.

command state: execute You do not have to type the name of the file that you wish to execute, simply click on the file you want and then press the **return** key.

5 The Marvel Objectbase

This section describes some important details of how MARVEL objectbases work in general, rather then referring to a specific environment such as *C/Marvel.* It should provide a good background for understanding much of the rest of this manual. We first discuss classes, then objects, then attributes.

The following discussion will be more clear if the user has the notion of *object oriented programming* in mind.

5.1 Classes

Classes are templates that are used to define the various kinds of objects you might want to deal with in your objectbase. These templates are used to define the structure of the objectbase, via two methods: *composite objects*, or the hierarchy we have been discussing thus far in the manual, and *links* or directed pointers to specific objects or attributes of objects.

Classes contain attributes that are used as templates when in comes time to create an object of a particular class.

Classes also define inheritance and subclassing amongst objects.

5.2 Objects

Objects are instantiations of classes. An object is the representation of an actual entity, such as a file. This representation includes *attributes* that define states, hierarchy and links. In MARVEL , an object has a file system component (which might be the contents of some file) and a set of attributes.

5.3 Attributes

Attributes are used to describe objects. There are a wide variety of types of attributes, including:

integer Normal definition of integers;

boolean Either **true** or **false**;

string Any combination of letters, numbers, dashes and underscores;

real Numbers with exponents, decimals, etc. They are treated as doubles;

enumerated A set of strings, where the value of the attribute is a member of the set;

user A user on the system. Can be the distingushed string **CurrentUser** to represent the current user of the system;

time The time. Can be the distingushed time **CurrentTime** to represent the creation time of the object this attribute belongs to;

text A text file. Not currently fully supported;

binary A binary file. Not currently fully supported;

set A set. Currently, only sets of other objects are supported, rather then sets of any attribute.

link A link to an other object, an other attribute, a set of objects, and so forth. Can also be a set of links. Can also be a generic link to anything. The link mechanism is very general.

6 Browsing through the objectbase

This section is an overview of various ways that you can browse through the objectbase. Only the Xwindows interface allows you to do browsing, the command line interface does not. Browsing is the idea of moving around your objectbase to gather information about the various objects that you have created.

6.1 Browse

Browse, which is located between **print** and **add** in the menu, has four options: **zoomin**, **zoomout**, **pan**, and **info**. These options allow you to change the graphic display of your tree and gain information about the place in the hierarchy that any of your objects occupies.

1. **Zoomin** will shrink the display of your objectbase by taking the object that you choose making it the root. The tree is resized to fill the **display window**.
2. **Zoomout** will enlarge the display by one level of hierarchy, giving the current root's parent(s) as the new root(s). The tree is resized to fill the **display window**.
3. **Pan** will examine the node of your choice. You will be queried:

   Pan right or left?

   the **kernel commands** will change to **right** and **left** and simply click on the one you prefer; the object to the right or left of the one from which you are panning are shown to you. The tree is resized to fill the **display window**.
4. **Info** will provide you with more information on the instance that you pick. You will be provided with the object's name, the owner attribute, the owner class, the parent object, and the type of the name.
5. **Done** exits **browse** and returns you to the original **kernel menu**.

6.2 Change

Change allows you to move around your tree and forest. Since many actions you do effect the current object (which is always shown in the **status window**) being able to alter the current object is obviously advantageous. **Change** provides the ability to jump around rather than having to climb the tree object by object.

To use **Change**:

1. select the **Change** option
2. read the **text window**, it should read

   Pick an object to change to.
3. select the object by clicking the pointer on it
4. if it is a valid object then the **Current Object** in the **status window** will be changed

6.3 Print

The very first command on the MARVEL menu is **print**. By choosing **print** your menu will offer you the following options:

all lists all the classes and their superclasses, attributes, objects and their attributes for all the instances. All the appropriate statuses of each object are listed. In addition, each child object is shown under it's parent in an easy to read fashion. This is done in a non-recursive manner.

class lists all the classes, their superclasses, and attributes currently in the data model.

inst lists the current data model using the class heading, so that all objects of the same class are grouped together. The status of each is listed along with the attributes.

rules prints information about one or all the rules in the objectbase. Either a specific rule name can be entered or by hitting the RETURN key all the rules will be listed.

rels is not currently available

string will search the class list for a match to the given string.

current lists the current object, it's place in the hierarchy (recursively) and it's attributes

graph is not currently available

6.4 Set Commands

The set command allows you to change various aspects of MARVEL to make it easier for you to work. For example, you can change the fonts of the commands to make them easier to read. Once you select the **set** command a new menu will appear in the **kernel command** menu. Unless it is otherwise noted, by simply choosing the options, its value will be changed from True to False or vice versa.

BoldFont By changing the font type the font in the **text** and **status** windows are altered.

SmallFont By changing the font type the font in the **display** window is altered.

NormFont By changing the font type the font in the menus is altered.

Verbose By changing **Verbose** to TRUE, you change the amount of information provided by the other **kernel** and **loaded** commands. With **Verbose** being TRUE, more information is provided; with **Verbose** set at FALSE only the absolute minimum information is provided.

AllInfo By changing **AllInfo** to True all the values of the attributes of the instantiated objects and attribute templates of classes of objects will be displayed. When it is False only the names of the objects in question will be printed.

AllMatches By setting **AllMatches** to True and giving it an object name all the objects of that name are printed in the **text** window.

AutoVal This should always be set to True because when it is set to False then the screen will not be updated when new objects are added or removed. The alterations will show only after a **print**.

DEBUG By setting **DEBUG** to True when an MSL strategy is loaded the loader provides information about the strategies begin loaded.

Depth By changing **Depth** the depth to which the objectbase can be printed can be altered; this requires a number to be given at the prompt.

RuleMode By setting to DWIM mode (Do What I Mean) or DWIT mode (Do What I Tell you) the intelligence of the rule overloading can be changed. DWIT is the default mode.

PrintLinks By setting **PrintLinks** to True, when in **print** the links between an object's attributes and other objects are shown.

Print BLinks By setting **Print BLinks** to True, when in **print** shows the objects linked to a particular object

DisplayLinks By choosing **DisplayLinks** and setting it to True the links will be displayed in the **display** window; setting it to False removes these links from the display

ShowAll This option will show you all the values for all the variables that exist in set.

7 Objectbase Manipulation Commands

This section explains to you the various commands that allow you to manipulate the objectbase. These manipulations include such things as adding objects, moving them, and copying them. While these commands are found both in the Xwindows Interface and the Command Line Interface, this section only explains the Xwindows Interface.

7.1 Add

The **add** command allows you to add objects. There are two ways to add instances to the objectbase, either hierarchically or horizontally. Adding hierarchically means that you are adding more instances of classes one level below the one in which you are currently working. Adding horizontally implies that you are inserting more instances of the current class. Consider the objectbase like a forest; when you add hierarchically you are extending the trunk and when you add horizontally you are growing a branch.

7.2 Link

The **link** command allows you to connect objects that are not connected in the hierarchy, but that you realize will need to use each other. For example, in a C program the "include" files. There are four types of links: **generic**, **typed**, **single**, and **multiple**. **Generic links** link any object or attribute to any object or attribute. **Typed links** link a specific type of object or attribute to a matching type of object or attribute. **Single links** link one object or attribute to only one other object or attribute. **Multiple links** link an object or attribute to numerous objects or attributes; this is the default.

7.3 Copy

Copy, the eighth box from the top, allows you to copy on object, it's children, and it's attributes from one place to some place else. By clicking on copy you are prompted in the **text** window with

Pick an object to copy, or pick done.

Should you decide that you do not want to copy anything you can escape from this command with the option **done**. To choose an object to copy click on it and you will then be prompted with

Pick an object to copy to, or pick done.

Here you choose the object that will be the parent to the object that you are copying. When you click on the parent object there will be a brief pause while the system savepoints the objectbase and then does the copying and finally in the **text window** you will see that the calculating is done. At the completion of the calculating the new version of the objectbase is shown on the screen with the appropriate tree structure. When you have finished all of the copying that you have to do then choose the **done** option in the **kernel commands**.

Remember that the object that you copy must be lower on the hierarchy than the object that will its parent(s). This relates back to the idea that you are copying the attributes as well as the object and so the attributes must fit where the object will be placed.

7.4 Move

Move, which is found directly below **copy**, is very similar to **copy**. Again you must choose an object, this time to move, and a parent object to which to move this child object. MARVEL will then unlink the object and all of its children and link it with its children to the chosen destination. When you are done moving should there be more than two objects of the same name which are of the same depth and children of the same parent; however, you will see that the name one of them will be slightly altered so that you and the system can tell the difference. Again, the parent object must be higher on the hierarchy than the object which you are moving.

The first prompt to appear in the **text window** after choosing the **move** option is

Pick an object to move, or pick done.

Simply, click the mouse pointer on the object that you wish to move. You will then be prompted with

Pick an object to move to, or pick done.

Again, click on the object to which you wish to move. When you have finished all the moving that you wish to click on the **done** option and you will exit.

7.5 Join

This command directly below **move** also reshapes your hierarchy. When you join two objects they must be of the same class. If they are not then you will be notified with a message saying that the joining has failed. The source object, the one that is being joined to other objects, will disappear and it's children will become part of the destination object. If you join two objects and some or all of their children have the same name then the set being moved, that is one set of children will be renamed. For example: if you have two children of different parents both named fred and you decide to join the parents then one of the children will become fred0 and the other will remain fred.

7.6 Rename

This command permits you to rename your objects. Simply choose the option and in the **text window** will appear the statement:

Pick an object to rename, or pick done.

Place the mouse pointer on the object that you wish renamed and click. In response to the request in the **text** window

Enter new name for < *objectname* >**:**

type

main.c.

The change will be made and you can continue altering names or exit by choosing done.

7.7 Delete

This command allows you to delete an object. You should only delete children that are not parents. If you delete parents then you will also recursively delete the children of that parent. Again, **delete** is fairly simple to use. Choose the **delete** option and

Pick an object to delete, or pick done.

Click on the object that you want to delete and the **text window** will query you as to whether or not you really want to delete this object. This second chance offers the option of canceling your delete before something important disappears. When you have finished all your deletions choose the **done** option and you will have exited delete.

8 Changing Strategies

This chapter explains the various ways in which the user can change the strategies that they are using. Without the mechanism of the strategies MARVEL would be useful for only one specific environment; the ability to change strategies provides MARVEL with a more generic use.

8.1 Loading Strategies

MARVEL's behavior can be changed by loading a new set of strategies that redefine the existing rules, and/or add new rules, or delete old rules. These strategies are either written by the local MARVEL administrator or provided as examples with the distribution tape. When you load a strategy there are three important things to remember:

1. The new strategies are loaded without regard to the current strategies.
2. **Load** recursively loads all the strategies that it imports, meaning that an entire environment call be loaded by loading the head strategy that imports all the needed strategies.
3. **Load** saves all the current strategies in a temporary file in the data directory of the current object.

Load is rather straight forward. Click on the **load** box. In the **text window** you will be prompted with

Enter strategy name:

You respond with the name of the strategy that you will be using, ask your supervisor if you are unsure. For our example we loaded **cmarvel_chaining**.

8.2 Merging Strategies

MARVEL also provides a merge command to merge a strategy in with the existing strategies. It is very similar to **load** except that it merges the strategy with the current set of strategies, rather than saving the current strategies and entering a whole new set. In that case MARVEL will notice the strategy was already loaded, and unload it before merging in the new version. **Merge**, like **load** is recursive and, so, will import all the strategies it needs. **Merge** becomes useful when you want to add a specific rule to your current strategies.

8.3 Unloading Strategies

Unload removes the strategies that you request. When you unload a strategy remember that the strategies that use the one which you are unloading are unloaded as well. You will be queried as to whether or not you really want to unload all of the strategies which MARVEL determines need to be unloaded in order to properly unload the strategy originally specified.

9 Advanced Features

This section contains a variety of features not covered in the tutorial you have now presumably read. These features include envelopes, savepointing, hardware platforms required, and a list of unsupported features and bugs.

9.1 Envelopes for Tool Invocation

MARVEL is a kernel that can be instantiated with the specific information applicable to a project. It does not provide a set of tools, but instead allows the user to use any tools available on Unix. These tools do not know anything about the structure and organization of the objectbase in which the project components are stored. Envelopes serve as an intermediary that map the inputs and outputs of tools to/from the objectbase. Let us take the compiler as an example. We assume that we are using the C compiler cc and that the C program that needs to be compiled is stored in a MARVEL objectbase similar to the one we have described. It is clear that there is a need for an envelope to call cc with all the appropriate arguments, and possibly do some postprocessing on the results. Finally, the envelope must inform MARVEL of the results of the compilation.

Envelopes can be written in you favorite Unix shell programming language (such as /bin/ksh or /bin/csh), or they can be local or COTS (Commercial Off The Shelf) tools. Thus, the envelope mechanism of MARVEL is quite simple, yet very powerful. Envelopes get information from MARVEL via the parameters specified in a rule. They return information via return codes. A zero return code activates the rule's first postcondition, a one the rule's second postcondition, and so forth. Thus MARVEL's behavior can be controlled relative to the success or failure of an envelope.

9.2 Savepoints

MARVEL creates backup versions of the objectbase and the current set of loaded rules at appropriate points during a session. All the information MARVEL creates is in somewhat readable ascii files. This information is a merged conglomeration of the contents of the strategies and objectbase, thus is in a different format then the strategy files dealt with in the tutorial. All the files in question are created in the *data* directory of the current objectbase root. The working versions of the files are called *objectbase* and *strategy*. The *objectbase* file contains all the instances MARVEL knows about for the objectbase in question. The *strategy* file contains all the rules, relations and class definitions for the currently loaded set of strategies.

During a MARVEL session, these files are saved into temporary versions (checkpointed) at appropriate points. These points include:

- After every **add** (objectbase),
- At the beginning of every **load**, **merge**, or **unload** command (strategy),
- At the end of every **load**, **merge**, or **unload** command (strategy),
- After a **save** command (objectbase), and
- At quit time, if the user does not specify elsewise (objectbase and strategy).

The temporary objectbase files are called *obj.<process_id>_<seq>*, and the temporary strategy files are called *str.<process_id>_<seq>*, and for those at the end of load, merge or unload commands *str.<process_id>_<seq>_new.* <process_id> is the process id of MARVEL, and <seq> in a sequence number assigned in MARVEL to assure uniqueness of files.

In the case of a system crash, appropriate copies (the newest ones, for example) of these backup files can be moved to *objectbase* and *strategy* in order to avoid loss of work. This must be done with appropriate Unix commands before restarting MARVEL.

Savepointing is done frequently in MARVEL due to the current nature of the system. It is an academic system, and is not of "production quality", per se. This savepointing is very fast, and users should not find it bothersome.

9.3 Execute

This command executes a MARVEL script file. A script file consists of the first fifteen bytes being: **#!marvel script** and the rest being any combination of commands. The changes that have occurred after the execution will not be visible on the MARVEL screen until a **print all** has been activated.

9.4 Objectbase Locking

The objectbase being used in a MARVEL session is locked, to prevent other users to access and potentially corrupt it. Since this version of MARVEL is single user, this is the appropriate thing to do. Locks are removed upon exiting, and in general in the rare case of a coredump. Certain signals are difficult to catch, and if the database remains locked, the mechanism must be broken. The lock is created by writing the user's name in the `.marvel_id` file in the objectbase directory. Removing this name will unlock the objectbase. This should **never** be done while MARVEL is running, as Unix would be happy to allow two people to overwrite each other, and MARVEL does not protect against it.

10 Hardware Requirements

MARVEL runs on Sun3's and Sun4's running SunOS 4.0, IBM RT's running AIX 2.2.1, and various Dec machines running Ultrix (v. 3.1). It has been most extensively tested on the Suns to this point, but we will be working with it on the RT's soon. It runs best on machines with at least eight megabytes of main memory. The entire system requires somewhere in the vicinity of 25 megabytes of disk storage to start with. To run the X windows interface, a display terminal that runs the X-11 version 3 or 4 server is required. Such a terminal need not be on the machine on which MARVEL is executing.

11 Unimplemented Features and Bugs

The following list contains a variety of things that are currently either not implemented, or don't work properly. This list is incomplete and your additions would be appreciated; however, updates and corrects will not happen immediately.

1. This bug list is incomplete.
2. On the RT's with a 5081 monitor, the menus come out with a very light yellow color, for some unclear reason. To get around this, assuming you are using `aixwm`, set the background color (with the `set` option on the main menu) to something like light blue. Note that this is different then the `xsetroot -solid` command.
3. MARVEL will sometimes coredump and leave the objectbase locked upon startup if it can not find fonts for X-11, or has some other connection problem. To restart, remove the `.marvel_id` file in the objectbase in question, and make a new one.
4. If the loader finds a syntax error, it is not always so graceful in exiting. This does not effect the MARVEL process, so ignore messages such as:

   Loader: internal memory error. Bye.
5. Relations are not implemented. Do not use them.
6. The **tree** option of the **print** command does not do much.
7. MSL does not do as much semantic checking as it could/should during compilation of strategies.
8. The **merge** command is buggy, and the **unload** command is very buggy.
9. Comparison of links might not work in the evaluator.
10. If **DISPLAY_BLINKS** is set, erroneous links (if any links exist) are displayed when anything but the full objectbase is in the display window.
11. The output of usage is different than the manual pages. Trust the manual pages first.
12. A few of the rules in *C/Marvel* do not do much.
13. Any environments included in the release except *C/Marvel* were developed with earlier versions of MARVEL, and are thus probably incompatible. The changes include minor grammar revisions, and changes in the internal storage format for the objectbase and rules currently loaded. We have just not had the time to update them.

14. This manual is not complete, the target finishing date is May, 1990. Thus it might not be as accurate as it should be.

We would be overjoyed to get bug reports, but please do not expect updates in particular.

12 Marvel Strategy Language

This section describes the *Marvel Strategy Language*, or MSL. MSL is used to write formal descriptions of MARVEL environments. We have included an entire MARVEL environment, *CMarvel*, in fact, in Appendix A of this manual. Most all of the features of MSL are exploited in that example, you should check it for examples throughout this section.

MSL is compiled into MARVEL by invoking either the load or merge command. This is in actuality a separate process operating, then communicating the results back to MARVEL via a condensed ascii format. There is minimal syntax checking, the line in which the syntax error was found is specified, and the file from which it came. Parsing in a given file of MSL constructs stops after the first syntax error. Semantic errors are similarly noted, however some semantic errors are not caught until the evaluator sees them while executing a rule. Should you see an unfamiliar and unfriendly message appear from the evaluator, it is probably because of such a semantic error in a MSL rule definition (of course, you might have found a new bug ...).

You can only load or merge one file of MSL definitions at a time, but that file can import others, in a recursive fashion. In fact, the parser will complain if you try to load a rule before defining all the things that the rule uses.

MSL is parsed by a bottom up, reentrant parser designed with the tools Yacc and Lex. Lex is only used for tokenizing. The parser makes one partial pass through it's input to determine the parsing order of imported MSL definition files, and a second pass to actually parse the definitions. MSL definition files must end with the suffix .load. Related MARVEL commands load and merge add on this suffix when searching for files.

This is an advanced part of the manual. We present MSL only briefly here, some intuition is going to be necessary to complete the picture. If you have not read about MARVEL previously, you might not get the entire picture. Such reading is recommended.

We first discuss the lexical issues, and then give all the grammar rules the parser recognizes. The discussion will make use of forms that `lex` and `yacc` understand. Anyone who understands simple regular expressions and BNF style rules should be able to figure them out.

12.1 Lexical Analysis

12.1.1 Keywords

The following keywords are reserved by MSL:

not	exists	forall	notexists
notforall	suchthat	rules	relations
end_relation	strategy	objectbase	imports
exports	and	or	string
integer	boolean	real	set_of
seq_of	single	user	CurrentTime
CurrentUser	time	link	any_att
any_inst	text	binary	enumerated
member	add	remove	true
false	superclass	end_objectbase	end

12.1.2 Numbers and Identifiers

Note that things in quotes are literal characters, — indicates options, + indicates 1 or more of the specified characters , and * indicates 0 or more of the specified characters. Items in parenthesis indicates groupings, and items in curly brackets indicate a predefined character class (see section a above).

Thus, the following definitions are used for numbers and identifiers:

```
DIGIT      [0-9]
LETTER     [a-zA-Z]
LETTERS    {LETTER}+
SPACES     [ \t]
IDSTRING   {LETTER}({LETTERS}|{DIGIT}+|_)*
COMMENT    #.*
```

The following fairly general forms of integers, real numbers and identifiers are then specified:

```
"-"{DIGIT}+ |
{DIGIT}+                         -- IVAL in grammar
("-"{DIGIT}*|{DIGIT}*)"."({DIGIT}+)  |
("-"{DIGIT}*|{DIGIT}*)"."({DIGIT}+)(E|e|E"-"|e"-"){DIGIT}
                                 -- RVAL in grammar
"?"{IDSTRING}                    -- VARIABLE in grammar
"?"{IDSTRING}"."{IDSTRING}  -- BVAR in grammar
"?"{IDSTRING}":"{IDSTRING}  -- PARAM in grammar
{IDSTRING}("/"{IDSTRING})+  -- PATH in grammar
{IDSTRING}                       -- ID in grammar
```

12.1.3 Other Symbols

These special symbols are used by MSL:

```
( ) { } [ ] : ; , .
=     -- EQ
<>    -- NEQ
<=    -- GEQ
>=    -- LEQ
::    -- D_COLON
```

12.2 The Grammar

The non-terminal **start** must start each file of MSL definitions. This section is organized by logical parts of the input.

```
start:       STRATEGY_kw ID imp_exp objbase rel_section rule_section
```

12.3 Imports and Exports

```
imp_exp:     IMPORTS_kw imp_name_list ';' EXPORTS_kw exp_name_list ';'

imp_name_list: /* nothing */
             | ID
             | imp_name_list ',' ID

exp_name_list: /* nothing */
             | ID
             | exp_name_list ',' ID
```

12.4 Class Definitions

```
objbase:     /* nothing */
             | OBJECTBASE_kw classes ENDOBJBASE_kw

classes:     class
             | classes class

class: ID D_COLON superclasses attributes END_kw
```

12.4.1 Inheritance and Specialization

```
superclasses: SUPERCLASS_kw ';'
            | SUPERCLASS_kw super_name_list ';'

super_name_list: ID
            | super_name_list ',' ID
```

12.4.2 Attributes

```
attributes:  attrib
            | attributes attrib

attrib:      ID ':' noninitable_type ';'
            | ID ':' autoinitable_type ';'
            | ID ':' initable_type ';'
            | ID ':' initable_type init_val ';'

autoinitable_type: USER_kw
            | TIME_kw

initable_type: STRING_kw
            | INTEGER_kw
            | REAL_kw
            | BOOLEAN_kw
            | enumerated_type

noninitable_type: file_type
            | complex_type
            | link_type

enumerated_type: '(' et_name_list ')'

file_type:   TEXT_kw
            | BINARY_kw

complex_type: SETOF_kw initable_type
            | SEQOF_kw initable_type
            | SETOF_kw file_type
            | SEQOF_kw file_type
            | SETOF_kw ID
            | SEQOF_kw ID
```

```
link_type:   LINK_kw
             | LINK_kw link_t
             | LINK_kw SINGLE_kw
             | LINK_kw link_t SINGLE_kw

link_t:      INTEGER_kw
             | INTEGER_kw
             | REAL_kw
             | BOOLEAN_kw
             | ENUMERATED_kw
             | USER_kw
             | TIME_kw
             | STRING_kw
             | TEXT_kw
             | BINARY_kw
             | LINK_kw
             | ID_kw
             | SETOF_kw ID
             | ANYATTRIBUTE_kw
             | ANYINSTANCE_kw


et_name_list: ID
             | et_name_list ',' ID

init_val:    /* nothing */
             | EQ_tok ID
             | EQ_tok PATH
             | EQ_tok TRUE_tok
             | EQ_tok FALSE_tok
             | EQ_tok IVAL
             | EQ_tok RVAL
```

12.5 Relations

Note that relations are not really supported in MARVEL currently. Typed links are available, however.

```
rel_section: /* nothing */
             | RELATIONS_kw relations
             ;
```

```
relations:    rel
              | relations rel

rel:          ID ':' rel_decl rel_fields ENDRELATION_kw

rel_decl:     ID type init_val ';'

rel_fields:   rel_decl
              | rel_fields rel_decl
```

12.6 Rules

```
rule_section: /* nothing */
              | RULES_kw rules

rules:        rule
              | rules rule

rule:     ID '[' parameters ']' ':' bindings ':'  precond actions mult_posts

parameters:   /* nothing */
              | PARAM
              | parameters ',' PARAM
```

12.6.1 Preconditions

```
bindings:     /* nothing */
              | binding
              | '(' AND_kw binding_list binding ')'

binding:      '(' binding_op ID VARIABLE SUCHTHAT_kw precond ')'

binding_list: binding
              | binding_list binding

binding_op:   EXISTS_kw
              | FORALL_kw
              | NOTEXISTS_kw
              | NOTFORALL_kw

precond:      /* nothing */
```

```
              | logical_expr

logical_expr_list: logical_expr
              | logical_expr_list logical_expr

logical_expr: '(' AND_kw logical_expr_list logical_expr ')'
              | '(' OR_kw logical_expr_list logical_expr ')'
              | '(' NOT_kw logical_expr ')'
              | '(' expression ')'
              | '(' set_expr ')'
```

12.6.2 Activities

```
actions:      '{' '}'
              | '{' actionlist '}'

actionlist:   action
              | actionlist action

action:       ID ID
              | ID ID act_var_list

act_var_list: VARIABLE
              | act_var_list VARIABLE
```

12.6.3 Postconditions

```
mult_posts:   ';'
              | mult_post_list

mult_post_list: post ';'
              | mult_post_list post ';'

post:         post_l_expr
              | '(' AND_kw post_l_expr_list post_l_expr ')'

post_l_expr_list: post_l_expr
              | post_l_expr_list post_l_expr

post_l_expr: '(' expression ')'
```

12.6.4 Expressions

The following rules for simple expressions are used for preconditions and postconditions.

```
expression:  BVAR string_exp_op string_expr
             | BVAR exp_op IVAL
             | BVAR exp_op RVAL

set_expr:    ID '[' ID ',' ID ']'
             | set_op '[' BVAR VARIABLE ']'

string_expr: BVAR
             | ID

exp_op:      string_exp_op
             TRUE_tok
             | FALSE_tok
             | GT_tok
             | LT_tok
             | GEQ_tok
             | LEQ_tok

string_exp_op: EQ_tok
             | NEQ_tok

set_op:      MEMBER_kw
             | ADD_kw
             | REMOVE_kw
```

13 Manual Pages

This section contains manual pages for all MARVEL kernel commands. The descriptions of the commands are designed to help you use the commands, rather than explain how they work. The explanations include the usage for both graphics interface and command line interface. These man pages are also located in MARVEL and can be accessed by using the **help** command.

Note that these are all the manual pages that come with the release system, local sites can add others which may not appear in this manual.

13.1 add

```
Marvel Help System          Built in command          Add

NAME:

    add -- add an object to the objectbase

USAGE:

  COMMAND LINE:

    add <object>
          Horizontally add <object> within the current class.
          Note that all but the top level class can have duplicate
          instances.

    add -a <attribute> <object>
          Vertically add <instance> downward to the specified
          <attribute> of the current instance.


  GRAPHICS:

    1.click on add
    2.click on either horiz or hier
    3.select the instance by typing in the attribute and the
      object name
    4.confirm the addition by clicking on the yes box, or deny the
      addition by clicking on the no box.

DESCRIPTION:

    Add adds a new object the objectbase.

    Marvel maintains a concept of a current object.  When adding
    an object, it is always relative to the system's current object.
    This enforces the structure of the system, as imposed by the
    current data model.  The current object can be changed with the
    change command.

    There are two basic ways to add an object within the Marvel
    system.  These are horizontally and vertically (hierarchically).

    A Marvel objectbase is a directed graph, with an embedded
```

hierarchy. Thus horizontal movement within the tree moves the current focus from one instance of a class to another. Vertical movement within the tree follows attributes of instantiated objects to an object of a different class. So you can view these movements as either in a downward direction or an upward direction. Often upward movements are looked at as movements towards more managerial objects, such as MODULES or PROJECTS, but this depends entirely upon how the current data model has been defined by the system administrator of Marvel.

EXAMPLES:

COMMAND LINE:

```
add new
```

Adds an object called new to the current class.

```
add -a modules NEW
```

Adds an instance called NEW and makes it a member of the modules attribute of the current object.

GRAPHICS:

1.click on add
2.click on horiz
3.type in "new"
4.click on yes

SEE ALSO:

change

13.2 change

```
Marvel Help System        Built in command              Change

NAME:

    change -- change to a different object in
              the objectbase

USAGE:

  COMMAND LINE:

    change
        print a detailed description of the current object.

    change <object>
        Horizontally change to <object>, within the current
        class.  Note that this can be ambiguous, and is thus
        not especially recommended.  It is provided for
        convenience.

    change -c <class> <object>
        Vertically change to <object>, within the class <class>.
        This command is very useful for upwards changes,
        especially to the root of a tree.  But note that it can
        be ambiguous, and is thus not especially recommended
        except as just described.

    change -a <attribute> <object>
        Vertically change downward to <object>, found by the
        specified <attribute> of the current object.

  GRAPHICS:

    1.click on change
    2.select the object by clicking on it or typing its name

DESCRIPTION:

    Change is used to change the system's current object.  The
    current object is important because it is used to decide
    what will be added in an add command.

    A Marvel objectbase is a directed graph, with an embedded
```

hierarchy. Thus horizontal movement within the tree moves the current focus from one instance of a class to another. Vertical movement within the tree follows attributes of instantiated objects to an object of a different class. So you can view these movements can be either in a downward direction or an upward direction. Often upward movements are looked at as movements towards more managerial objects, such as MODULES or PROJECTS, but this depends entirely upon how the current data model has been defined by the system administrator of Marvel.

Since a Marvel data model can contain objects with the same names, it is possible for some change commands other then those specifying an attribute of an instantiated object to be ambiguous. Such changes are allowed for ease of navigation through the database, but should be used with caution.

Change, without any parameters, will print where in the data model you currently are. Such a remainder can also be achieved with the verbose option set, which causes the prompt to be a terse description of the current location (see set).

EXAMPLES:

COMMAND LINE:

```
change -c projects PROJ
```

GRAPHICS:

1.click on change
2.click on proj

SEE ALSO:

add, set.

13.3 execute

```
Marvel Help System          Built in command              Execute

NAME:

    execute -- execute a Marvel script

USAGE:

    execute <file>

DESCRIPTION:

    The execute command is used to execute a Marvel command
    script.  To create such a script, put any combination of
    commands in a readable file.  The first 15 bytes of the
    file must be:

         #!marvel script

    After that, any commands can appear.

    Regardless of the interface, Marvel command scripts are
    executed as if in the line interface.  Thus all parameters
    to commands must appear.  Any further input required will
    be queried for exactly as in the line interface.

CAVEATS:

    In the graphics interface, the screen will not get redrawn
    before the script is finished, thus any changes to the
    Marvel objectbase will not visible until a "print all"
    command is issued.
```

13.4 help

```
Marvel Help System        Built in command             Help

NAME:

    help -- get help

USAGE:

    help command
    help subject
    help ?

DESCRIPTION:

    Specifying a command will give you help for that command.
    The full name to the command must be given.

    Specifying a subject will give you help for that subject.
    This tends to be documentary, rather then for answering
    specific questions.

    Specifying a '?' will give you a list of topics for which
    help is currently available.  These topics include all
    built in commands, help for rules, as provided by your local
    system administrator, and general subjects of help.

SEE ALSO:

    Marvel Tutorial, by Mara W. Cohen, Naser S. Barghouti and
    Michael H. Sokolsky.

    Marvel literature.
```

13.5 link

```
Marvel Help System        Built in command              Link

NAME:

    Link -- link to an instance or attribute of an instance.

USAGE:

  COMMAND LINE:

    link    <source-att> <dest-class> <dest-inst>
    link -a <source-att> <dest-class> <dest-inst> <dest-att>

  GRAPHICS:

    1.click on link
    2.click on
      a.instance
        i.click on the destination instance or type its name and press
          return
      b.attribute
        i.click on the destination instance or type its name and press
          return
        ii.click on the destination attribute listed in the command menu

DESCRIPTION:

    The link command is used to define a graphical link from
    the current instance to the destination instance, or
    optionally, to an attribute of the destination instance.
    The destination instance may be the current (source)
    instance.  The source attribute must of type "LINK".  Only
    one link is allowed directly from the source attribute to
    the destination instance, although there may also be links
    to as many unique destination attributes within that
    instance as required.

    This command will fail if any of the following are true:

    1. Any of the items specified by the parameters are not
       found in the objectbase.
```

2. The source attribute specified is not a LINK attribute.

3. The link already exists.

SEE ALSO:

unlink, set

13.6 load

```
Marvel Help System        Built in command            Load

NAME:

    load -- load a strategy

USAGE:

  COMMAND LINE:

    load <strategy_name>

  GRAPHICS:

    1.click on load
    2.type the strategy name and hit return

DESCRIPTION:

    Load loads the strategy <strategy_name>.load, which
    was written either in the MSL structure editor,
    or by your favorite text editor with great care.
    It is expected that any necessary path information
    will be supplied with the argument.

    Load does not take the current set of strategies
    which are loaded into consideration.  It just reloads
    regardless of the situation.  Load is recursive,
    in that it loads the strategy specified, and all
    strategies (recursively) which that strategy imports.
    Thus, one head strategy can load a complete environment.

    Load calls a separate program to do the loading.  This
    provides fault tolerance within Marvel.  If anything
    fails during the load, such as an imported strategy not
    being found, the command will have no effect, the user
    can find (or fix) the strategies in question, and start
    again.

    In reality, load loads the top level strategy, and
    merges all imported strategies with that one.  Thus,
    a uniform objectbase and collection of rules is
```

created.

Before beginning to do any work, load saves all the current strategies in a temporary file in the data directory of the current root directory. The file is called str.<pid>_<seq>. Pid is the process id of Marvel, and seq is a non-negative integer. Upon successful completion of the load, another file, with the same name but _new appended to the end is created. These files can be used to restore Marvel to a previous state after a crash.

A Unix environment variable called MARVEL_LOADER must be set prior to starting up Marvel. The system will not startup unless this variable points to a valid executable program.

EXAMPLES:

COMMAND LINE:

```
load strategy/cmarvel_chaining
```

loads the strategy called cmarvel_chaining in a directory called strategies under the current marvel objectbase. chaining, and the strategies it imports, will entirely comprise the environment, regardless of what was present at the onset of the command.

GRAPHICS:

1.click on load
2.type in cmarvel_chaining

SEE ALSO:

merge, unload.

13.7 marvel

```
Marvel Help System                                 Subject: Marvel

NAME:

  marvel

USAGE:

   marvel [objectbase]

DESCRIPTION:

   Marvel is a knowledge based software development environment.

   A marvel objectbase is a persistent object structure which
   maps to the file system, and a set of extensible rules which
   have been loaded into Marvel to create an environment.

SEE ALSO:

   Marvel Tutorial, by Mara W. Cohen, Naser S. Barghouti and
   Michael H. Sokolsky.
   Marvel literature.
```

13.8 merge

```
Marvel Help System        Built in command              Merge

NAME:

    merge -- merge a strategy into the system

USAGE:

  COMMAND LINE:

    merge <strategy name>

  GRAPHICS:

    1.click on merge
    2.type in the strategy name

DESCRIPTION:

    Merge merges the strategy found in <strategy>.load into
    the current set of strategies.  Merge is very similar
    to load, except that it uses the current state of
    Marvel to start with, and merges everything into that
    state.

    If Merge is told to merge a strategy which is already
    in the system, it will query the user to unload that
    strategy.  If the user says no, the merge fails. Other-
    wise, that strategy is unloaded, and the new copy is
    merged back in.  This is very useful for making changes
    to a strategy, because you do not need to explicitly unload
    the strategy before reloading it.

    It is important that merge, like load, is recursive in
    it's loading of imports.  This includes imports of
    already loaded strategies.  An important factor here
    is that the only strategy which merge will unload is the
    one which is already in the system at the top level.
    Users are warned to be careful when changing multiple
    strategies at once.  There are arguments for this, and
    for unloading all imported strategies which are not
    appropriate discussion for this manual page.
```

The load manual page contains further descriptions of the workings of load, all the implementation details there apply to merge.

EXAMPLES:

COMMAND LINE:

```
merge strategies/cmarvel_chaining
```

merges the strategy called chaining in a directory called strategies under the current marvel objectbase. The resultant environment is a combination of this new strategy, and what used to exist.

GRAPHICS:

1.click on merge
2.type in cmarvel_chaining and hit return

SEE ALSO:

load, unload.

13.9 print

```
Marvel Help System        Built in command            Print

NAME:

    print -- print information about the objectbase
             and rules.

USAGE:

  COMMAND LINE:

    print -r <rule_name>
    print -R <relation_name>

        or

    print -a
    print -c
    print -i
    print <class>
    print <class> <instance> <attribute> <instance>
    print <class> <instance> <attribute> <instance> <attribute>

  GRAPHICS:

    1.click on print
    2.click on
      a.all
      b.class
      c.inst
      d.rules
      e.rels
      f.string
      g.current
      h.graph

 DESCRIPTION:

    Print is used to get information about the current
    instantiation of Marvel, and to browse the Marvel object-
    base.
```

Print -r, in the Command Line, and the option all, in the Graphics, prints all the rules currently in the system. A terse description is given. Print -r <rule_name> prints a detailed description of <rule_name>, showing preconditions required for the rule to be executed and postconditions set upon success or failure of the rule. Furthermore, descriptions of where the rule will chain to (either forwards or backwards) are shown for preconditions and postconditions.

Print -R ,in the Command Line, and the option rels, in the Graphics, prints all the relations currently in the system. (It does not currently work in the Graphics.) By specifying <relation_name> prints out information for that specific relation.

The remainder of the options to print control printing of information relating to the objectbase. What gets printed depends highly upon the print control variables allmatches, allinfo and depth. See set for a complete description of the functionality of these variables.

With no arguments, print, in the Command Line, and current, in the Graphics, prints the current instance, and any hierarchical information which can be derived from the data model.

Print -a, in the Command Line, and the option all, in the Graphics, prints useful information about the entire data model. For each class, attributes, and instances with attribute values are printed.

Print -c, in the Command Line, and the option class, in the Graphics, prints information about a specified class. Included are all the attributes and instances of the specified class, as in print -a.

Print -i, in the Command Line, and the option inst, in the Graphics, prints a listing of instances of classes. This option is especially helpful when allinfo is FALSE (see set), as a terse list of the contents of the objectbase is output.

The remainder of the options describe either a particular class, instance, or attribute, respectively. Anything can be specified because of the recursive capability described in the usage. This facility allows full browsing capability.

SEE ALSO:

set.

13.10 prompt

```
Marvel Help System          Built in command              Prompt

NAME:

    prompt -- change the system prompt.

USAGE:

  COMMAND LINE:

    prompt <new_prompt>

  GRAPHICS:

    This does not exist in the Graphics interface because prompts
    are not used.

DESCRIPTION:

    The prompt command is used to change the system prompt
    from "marvel:" to something else.

    If verbose (see set) is TRUE, the prompt is set every time
    a change command is issued, thus this command will have
    limited purpose.

EXAMPLE:

    prompt marvel_is_great:

    sets the marvel command prompt to the string
    "marvel_is_great:"

SEE ALSO:

    set.
```

13.11 quit

```
Marvel Help System        Built in command         Quit

NAME:

    quit -- quit Marvel

USAGE:

  COMMAND LINE:

    quit
    quit -s

  GRAPHICS:

    1.click on quit
    2.click on
      a.save
      b.no save
      c.cancel

DESCRIPTION:

    Quit is the proper method of leaving a session with Marvel.

    Quit queries the user to save the current state of the
    system, this includes the objectbase and the environment.
    In general, this question should be answered with a "y",
    in order to continue with Marvel at a later point.

    In the Command Line interface, -s can be specified in order
    to quit and save, without being queried about really quitting.

    In the Graphics interface, the save option saves the data model
    when you quit, the no save option does not save the data model
    when you quit, and the cancel option lets you return to Marvel
    without quitting.

SEE ALSO:

    save.
```

13.12 readob

```
Marvel Help System        Built in command            Readob

NAME:

    readob -- reread the objectbase.

USAGE:

  COMMAND LINE:

    readob

  GRAPHICS:

    This command does not work in the graphics interface.

DESCRIPTION:

    readob is used to read in the objectbase when an initial
    read has failed due to a class definition not being
    present in the current environment.

    The unwary user should be aware that this command can
    have dangerous implications, and therefore should not be
    used indiscriminately.

SEE ALSO:

    save
```

13.13 save

```
Marvel Help System          Built in command               Save

NAME:

    save -- write out the objectbase or strategies.

USAGE:

  COMMAND LINE:

    save [-o] [-s]

  GRAPHICS:

    1.click on save
    2.click on
      a.save - to save the objectbase
      b.cancel - to exit the save command without saving

DESCRIPTION:

    Save saves the current state of the objectbase.  It is
    recommended that save be executed frequently, to assure
    a consistent database.

    Save -s saves the current environment description.  This
    is generally not necessary, as it is done automatically
    at quit time without user intervention.  After loading,
    unloading and merging a number of strategies, however,
    it is prudent to issue save -s before continuing on with
    a lot more work within that new environment.

    -o is the default, but must be specified with -s to
    save both the objectbase and environment at the same time.

SEE ALSO:

    quit.
```

13.14 set

```
Marvel Help System        Built in command              Set

NAME:

    set -- set certain system parameters.

USAGE:

  COMMAND LINE:

    set [allmatches] [allinfo] [verbose]
       [depth <number>] [lines <number>]
       [DEBUG] [PARSE_DEBUG]

  GRAPHICS:

    1.click on set
    2.click on the variable that you wish to change
    3.if you choose
      a.DisplayLinks - the links will be shown and nothing will be
        changed
      b.ShowAll - the values of the variables will be listed in the
        Text window

DESCRIPTION:

    Set sets up various characteristerics for Marvel to use.
    Without any arguments, set prints all the Marvel variables,
    and their current value.  As many variables as is desired
    can be set in one single Marvel command. Set echo's out the
    new values of variables that are set.  For those variables
    which do not require a <number> parameter, set reverses
    the sense of the variable, which can be either TRUE or
    FALSE.  The default for all these variables is FALSE.

    Allmatches is used when printing the objectbase.  When a
    user prints an instance of a class, it's name need not be
    unique for that class.  If allmatches is TRUE, the system
    will print all matches of an instance, rather then just the
    default of the first match.
```

Allinfo is used when printing the objectbase. If allinfo is TRUE, all values of attributes of instantiated objects will be displayed, as well as all template attributes of classes of objects. Otherwise, just the names of the objects in question will be printed.

Verbose is used to control a few friendly aspects of Marvel. With verbose set to TRUE, the marvel prompt will always reflect Marvel's current location within the objectbase. Also, users will be prompted with a query when adding instances to be certain that is what they desire to do. Verbose is highly recommended for beginners.

Depth is used when printing the objectbase. When appropriate, the objectbase can be printed recursively to a depth specified by depth. If depth is a large number, the entire objectbase below a specified instance will be recursively printed. The default value of depth is 0.

Lines is used when printing the objectbase. It should be set to the number of lines in the display terminal, as it controls page at a time output. The default value of lines is 24.

SEE ALSO:

print

13.15 unlink

```
Marvel Help System        Built in command            Unlink

NAME:

    Unlink -- remove a link created previously using LINK.

USAGE:

  COMMAND LINE:

    unlink    <source-att> <dest-class> <dest-inst>
    unlink -a <source-att> <dest-class> <dest-inst> <dest-att>

  GRAPHICS:
    1.click on unlink
    2.click on
      a.instance
        i.click on the destination instance or type its name and
          hit return
      b.attribute
        i.click on the destination attribute or type its name and
          hit return
        ii.click on the attribute in the command menu

DESCRIPTION:

    The unlink command is used to permanently remove a
    graphical link which exists from the current instance to
    the destination instance or attribute.  The destination
    instance may be the current (source) instance.  The source
    attribute must of type "LINK".

    Unlink will fail if either of the following are true:

    1) The specified source attribute is not a LINK attribute.

    2) The link does not exist.


SEE ALSO:
```

```
link, set
```

13.16 unload

```
Marvel Help System        Built in command             Unload

NAME:

   unload -- unload a strategy

USAGE:

  COMMAND LINE:

   unload <strategy name>

  GRAPHICS:

   1.click on unload
   2.type in the strategy name
   3.respond yes to the query

DESCRIPTION:

   Unload unloads <strategy name>.  In order to keep the
   system consistent, it checks all the other strategies
   recursively to see if this strategy is used.  If so,
   those strategies are also unloaded.  After a list of
   strategies to be unloaded, the user is queried with
   this list, just in case the magnitude of the unload was
   not desired.

   The load manual page contains further descriptions of
   the workings of load, all the implementation details
   there apply to unload.

EXAMPLES:

  COMMAND LINE:

   unload strategies/chaining

      unloads the strategy called chaining in a directory
      called strategies under the current marvel objectbase.
      All other strategies which import chaining will get
      unloaded also.
```

```
SEE ALSO:

    load, merge.
```

13.17 usage

```
Marvel Help System        Built in command              Usage

NAME:

    usage -- find the usage of a command

USAGE:

  COMMAND LINE:

    usage command
    usage *

  GRAPHICS:

    1.click on usage
    2.click on
     a.all
     b.command
       i.click on a command

DESCRIPTION:

    The usage command is used to find out how to use a marvel
    built in command.  Currently, there are no exactly similar
    facilities for rules.  To determine how to use a rule,
    type

        "print -r <rule name>"

    Specifying command will print usage for that command.
    Specifying '*' will print usage for all the commands
    currently in the system.

SEE ALSO:

    help, print.
```

13.18 ?

```
Marvel Help System        Built in command                  ?

NAME:

    ?

USAGE:

  COMMAND LINE:

    ?

  GRAPHICS:
    1.click on ?

 DESCRIPTION:

    The ? command prints a list of available commands and rules
    at the current time.

 SEE ALSO:

    help.
```

A CMarvel Strategy Files

This section contains all the strategy files for the C/MARVEL environment discussed in this manual. There is one file of objectbase class definitions, a file for each of the major categories of tools supported by C/MARVEL, and a root file that imports all the others (see 8). Appendix B contains the envelopes used by the rules following.

A.1 cmarvel_chaining.load

```
#
#                    Marvel Software Development Environment
#
#                                Copyright 1990
#                     The Trustees of Columbia University
#                             in the City of New York
#                              All Rights Reserved
#

# This file contains MSL commands to build an environment for developing
# C programs using a maximal amount of chaining amongst the rules.  In
# addition, all the rules available in cmarvel are contained here.  If
# a user only needed a subset of these rules, an master file similar to
# this one could be created that contains just the appropriate subset.

strategy cmarvel_chaining

# Import all the addition data and process models needed to build up the
# environment.  Order is important here.

imports data_model, build, archive, compile, debug, edit, rcs;
exports all;
```

A.2 data_model.load

```
#
#                    Marvel Software Development Environment
#
#                                Copyright 1990
#                     The Trustees of Columbia University
#                              in the City of New York
#                               All Rights Reserved
#

strategy data_model

# This strategy contains all the class definitions needed for a typical
# C environment.  The class definitions are imported by all other
# strategies that define various aspects of the process model for
# C/Marvel.

# Interface with other strategies.  Since this is a basic data model that
# all other strategies import, we don't specify anything.

imports none;
exports all;

# Class definitions

objectbase

# GROUP is the top-level class.  An instance of GROUP contains several
# projects.  The fact that it is top level is set in the user's
# environment as part of the startup of Marvel.  So a Marvel objectbase
# can contain several group objects.

GROUP :: superclass ENTITY;
    name : string;
    status : (Active,NotActive) = Active;
    projects : set_of PROJECT;
end



# PROJECT is an entity that defines much of the structure of a typical
# software project.  PROJECTs can contain libraries, binaries, programs
# documents and includes in this example.
```

```
PROJECT :: superclass ENTITY;
    name : string;
    status : (Release,Maintenance,Development) = Development;
    archive_status : (Archived,NotArchived) = NotArchived;
    build_status : (Allbuilt,NotBuilt) = NotBuilt;
    libraries : set_of LIB;
    bin : set_of BIN;
    programs : set_of PROGRAM;
    doc : set_of DOC;
    incs : set_of INC;
end




# PROGRAM is important to distinguish from PROJECT.  A PROGRAM is a single
# executable unit, whereas a PROJECT is a collection of PROGRAMs, and other
# entities.  PROGRAMs thus contains things like documents, cfiles, modules,
# if it is large, and include files.

PROGRAM :: superclass ENTITY;
    name : string;
    build_status : (Built,NotBuilt,Error) = NotBuilt;
    debug_status : (OK,NeedsDebugging) = OK;
    docs : set_of DOC;
    cfiles : set_of CFILE;
    modules : set_of MODULE;
    lany_att : link any_att;
    lany_inst : link any_inst;
    lset_of_cfile : link set_of CFILE;
    lcfile : link CFILE;
    ltime : link time;
    lsingle_time : link time single;
    main : link;
    incs : set_of INC;
    ofile : binary;
end




# LIB is a shared archive type library.  It consists of modules, which in
# turn contain c files.  The ultimate representation of a library is a
# .a file, that is, an archive format file.

LIB :: superclass ENTITY;
```

```
    name : string;
    status : (Uptodate,NotUptodate) = NotUptodate;
    archive_status : (Archived,NotArchived) = NotArchived;
    modules : set_of MODULE;
    programs : link;
    bfile : binary;
end



# Typically, Libraries contain several organizational MODULEs, each of which
# contain .c and possibly .h files.

MODULE :: superclass ENTITY;
    name : string;
    status : (Archived,NotArchived,Error) = NotArchived;
    cfiles : set_of CFILE;
    hfiles : set_of HFILE;
    archive : link;
end



# FILE is the generic class for anything that is represented as a unix
# file.  There are specializations (subtypes) for CFILE, HFILE and DOCFILE
# in this system.

FILE :: superclass ENTITY;
    name : string;
    owner : user;
    timestamp : time;
    reservation_status : (Checked_out,Available,Error) = Available;
    version : text;
    contents : text;
end



# Extra information is needed to record the state of compilation and
# analysis (lint, in our case) for CFILEs.

CFILE :: superclass FILE;
    compile_status : (Compiled,NotCompiled,Error) = Error;
    analyze_status : (Analyzed,NotAnalyzed,Error) = Error;
    documentation : link;
end




# For HFILEs, we only want to know if they have been modified recently,
# which will cause a global recompilation.

HFILE :: superclass FILE;
    recompile_mod : (Yes,No) = Yes;
end




# For DOCFILEs, we only want to know if they have been reformatted recently,
# so we can reformat the document.

DOCFILE :: superclass FILE;
    reformat_doc : (Yes,No) = Yes;
end




# DOC is a class that represents an entire set of documents, typically for
# a PROJECT or PROGRAM.  A DOC can contain individual documents, and files
# of it's own.

DOC :: superclass ENTITY;
    name : string;
    type : (Latex, Scribe, Nroff) = Latex;
    files : set_of DOCFILE;
    documents : set_of DOCUMENT;
end




# DOCUMENT represents a complete individual document, such as a user's manual
# or technical report.

DOCUMENT :: superclass ENTITY;
    name : string;
    files : set_of DOCFILE;
    formated_copy : binary;
end



# INC represents a set of include (.h) files.

INC :: superclass ENTITY;
    name : string;
    hfiles : set_of HFILE;
end



# BIN represents a place where binaries for PROGRAMs (parts of a PROJECT) are
# kept.

BIN :: superclass ENTITY;
    name : string;
    executable : binary;
    program : link;
    project : link;
    archive : link;
end

end_objectbase
```

A.3 edit.load

```
#
#                    Marvel Software Development Environment
#
#                                Copyright 1990
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#

strategy edit

# This strategy defines the editor tool and a viewer tool that can display
# either the contents of a text file or the errors associated with a
# particular C file.  The rules for editing are overloaded, they set
# appropriate attributes depending upon the type of object being edited.

imports data_model, rcs;
exports all;

objectbase

EDITOR :: superclass TOOL;
    edit : string = editor;
    multi_edit : string = multi_editor;
end

VIEWER :: superclass TOOL;
    viewer : string = viewer;
    viewCerr : string = viewCerr;
    viewAerr : string = viewAerr;
end

end_objectbase


rules

# this edit rule is for editing document files.

edit[?f:DOCFILE]:

    # if the file has been reserved, you can go ahead and edit it
    :
    (?f.reservation_status = Checked_out)

    { EDITOR edit ?f }

    (and (?f.reformat_doc = Yes)(?f.timestamp = CurrentTime));



# this edit rule is for editing include files.

edit[?f:HFILE]:

    # if the file has been reserved, you can go ahead and edit it
    :
    (?f.reservation_status = Checked_out)

    { EDITOR edit ?f }

    (and (?f.recompile_mod = Yes)(?f.timestamp = CurrentTime));



# this edit rule is for editing c files.  Note that all these rules have the
# same activities, but different postconditions.  If there were special
# editors, they could be invoked by calling edit rules with different
# activities.

edit[?f:CFILE]:

    # if the file has been reserved, you can go ahead and edit it
    :
    (?f.reservation_status = Checked_out)

    { EDITOR edit ?f }

    (and (?f.compile_status = NotCompiled)
         (?f.analyze_status = NotAnalyzed)
         (?f.timestamp = CurrentTime));



#edit[?f:CFILE, ?q:CFILE]:
#
#    # if the file has been reserved, you can go ahead and edit it
#    :
```

```
#     (?f.reservation_status = Checked_out)
#
#     { EDITOR multi_edit ?f ?q}
#
#     (and (?f.compile_status = NotCompiled)
#          (?f.analyze_status = NotAnalyzed));
#

# this rule just views any kind of file with a pager, like more or less.

view[?f:FILE]:

    :

    { VIEWER viewer ?f }

;

# The following rules view output from the compiler and analyzer.

viewCerr[?f:CFILE]:

    :

    { VIEWER viewCerr ?f }

;


viewAerr[?f:CFILE]:

    :
    { VIEWER viewAerr ?f }

;
```

A.4 compile.load

```
#
#                    Marvel Software Development Environment
#
#                                Copyright 1990
#                      The Trustees of Columbia University
#                             in the City of New York
#                              All Rights Reserved
#

strategy compile

# This strategy contains rules to compile and analyze CFILE type objects.
# Compilation is done with cc, and analyzis with lint.  In our example,
# a file must successfully be analyzed before it is compiled.

imports data_model;
exports all;


objectbase


COMPILER :: superclass TOOL;
    compile : string = compile;
end


ANALYZER :: superclass TOOL;
    analyze : string = analyze;
end


end_objectbase


rules


compile [?f:CFILE]:

    # if the C file has been analyzed successfuly but not yet compiled,
    # you can compile it.  The compilation changes the status of the C
    # file to either compiled or error.
    :
    (and (?f.analyze_status = Analyzed)
         (?f.compile_status = NotCompiled))

    { COMPILER compile ?f }

    (?f.compile_status = Compiled);
    (?f.compile_status = Error);



analyze[?f:CFILE]:

    :
    (?f.analyze_status = NotAnalyzed)

    { ANALYZER analyze ?f }

    (?f.analyze_status = Analyzed);
    (?f.analyze_status = Error);
```

A.5 archive.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1990
#                      The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#

# This strategy contains the definition of the archiver tool that can
# call either a script to archive a whole project, or a script to
# archive a module.

strategy archive

# All we need for this part of the process model to be loaded
# is the data model.

imports data_model;
exports all;


objectbase

ARCHIVER :: superclass TOOL;
    archive : string = archive;
    list_archive : string = list_archive;
end

end_objectbase


rules

# arch_proj: archive all the libraries in this project.  This rule in an
#            inference rule (one with an empty activity) that causes
#            arch_lib to be executed (via chaining) for all the libraries
#            in the PROJECT.

arch_proj[?proj:PROJECT]:

    (forall LIB ?l suchthat (member [?proj.libraries ?l]))
    :
    (?l.archive_status = Archived)

    { }

    (?proj.archive_status = Archived);
    (?proj.archive_status = Error);




# arch_lib: archive all the modules in each library.  Again, this is an
#           inference rule that causes arch_mod to be executed to do the
#           real work.

arch_lib[?l:LIB]:

    (forall MODULE ?mod suchthat (member [?l.modules ?mod]))
    :
    (?mod.status = Archived)

    { }

    (?l.archive_status = Archived);
    (?l.archive_status = NotArchived);




# arch_mod:  This rule archives all the cfiles in a project, after they
#            have been analyzed and compiled.

arch_mod[?m:MODULE]:

    (forall CFILE ?f suchthat (member [?m.cfiles ?f]))
    :
    (and (?f.analyze_status = Analyzed)
         (?f.compile_status = Compiled))

    { ARCHIVER archive ?m }

    (?m.status = Archived);
    (?m.status = NotArchived);
```

```
# list_arch: this rule just lists the contents of an archive.

list_arch[?1:LIB]:

    :

    { ARCHIVER list_archive ?1 }

    ;
```

A.6 build.load

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1990
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#

strategy build

# This strategy defines 2 tool objects, and three rules to access these
# tool objects.  The BUILD tool has two alternative operations: either
# build_project, to build the entire project, or build_program that builds
# one program of the project.

imports data_model;
exports all;


objectbase


RELEASE :: superclass TOOL;
    release : string = release;
end

BUILD :: superclass TOOL;
    build_program : string = build;
    build_project : string = all_build;
end


end_objectbase


rules


release [?proj:PROJECT]:

    # if all programs that belong to the project have been built, the
    # project can be released.  Otherwise, the project is either still
    # in maintenance or development.  The script that the RELEASE tool
    # executes finds out which phase the project is in.

    (forall PROGRAM ?p suchthat (and (member [?proj.programs ?p])
                                     (?proj.build_status = Allbuilt)))
    :
    (?p.build_status = Built)

    # call the release operation of the release tool with the project
    # as argument.

      { RELEASE release ?proj }

    # alternative postconditions :
    (?proj.status = Release) ;
    (?proj.status = Maintenance);
    (?proj.status = Development);


build_all [?proj:PROJECT]:

    # we still need the colon
    :
    # call routine that will build all the programs in a project

    { BUILD build_project ?proj }

    (?proj.build_status = AllBuilt);
    (?proj.build_status = NotBuilt);



build[?p:PROGRAM]:

    # if all the C files of the program have been successfully
    # analyzed and compiled, then you can build the program

    (and (forall PROJECT ?P suchthat (member [?P.programs ?p]))
         (forall LIB ?l suchthat (member [?P.libraries ?l]))
         (forall MODULE ?m suchthat (member [?p.modules ?m]))
         (forall CFILE ?c suchthat (member [?p.cfiles ?c])))
    :
    (and (?c.analyze_status = Analyzed)
         (?c.compile_status = Compiled)
         (?m.status = Archived)
```

```
        (?1.archive_status = Archived))

    { BUILD build_program ?p }

    (?p.build_status = Built);
    (?p.build_status = NotBuilt);
```

A.7 debug.load

```
#
#                   Marvel Software Development Environment
#
#                              Copyright 1990
#                    The Trustees of Columbia University
#                           in the City of New York
#                             All Rights Reserved
#

strategy debug

# This strategy contains rules to debug and execute PROGRAM type objects.

imports data_model;
exports all;


objectbase


DEBUGGER :: superclass TOOL;
    debug : string = debug;
    exec : string = execute;
end

end_objectbase


rules


exec_prog[?p:PROGRAM]:

    :
    (?p.build_status = Built)

    { DEBUGGER exec ?p }

    (?p.debug_status = OK);
    (?p.debug_status = NeedsDebugging);
```

```
# deposit: deposit an object.  This rule works on the same objects as the
#          reserve rule.

deposit[?f:FILE]:

    :
    (and (?f.owner = CurrentUser)
         (?f.reservation_status = Checked_out))

    { RCS deposit ?f }

    (?f.reservation_status = Available);
    (?f.reservation_status = Checked_out);
```

B C/Marvel Envelopes

This section contains all the envelopes for the CMarvel environment discussed in this manual. The following envelopes are all /bin/sh scripts. They appear alphabetically.

B.1 all_arch

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1989
#                       The Trustees of Columbia University
#                               in the City of New York
#                                All Rights Reserved
#
# all_arch envelope
#
# usage: all_arch

echo all_arch $1 on 'date'

# If this envelope gets called, it is just a placeholder to assert a
# postcondition.

return 0
```

B.2 all_build

```
#
#                    Marvel Software Development Environment
#
#                              Copyright 1989
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#
# all_build envelope
#
# usage: all_build

echo all_build $1 on `date`

# If this envelope gets called, it is just a placeholder to assert a
# postcondition.

return 0
```

B.3 analyze

```
#!/bin/ksh

#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# analyze envelope
#
# usage: compile [CFILE]
#


cd $1
cfile=`basename $1`

echo "$0 $cfile on `date`"
echo

log=l_err

echo "$0 $1 on `date`" > $log
echo >> $log
echo >> $log

# we need to make the -I list

mod_or_prog=`dirname $1`
mod_or_prog=`dirname $mod_or_prog`

ifiles=`ls -d $mod_or_prog/hfiles/* $mod_or_prog/incs/*/hfiles/* 2>/dev/null`

idirs=
if [ "x$ifiles" != "x" ]
then
    for f in $ifiles
    do
        idirs="$idirs -I$f"
    done
fi
```

```
echo "lint $idirs $cfile" >> $log
lint $idirs $cfile >> $log 2>&1

# for now

if [ $? -eq 0 ]
then
    echo analysis successful, results available with viewAerr
    echo analysis successful >> $log
    exit 0
else
    echo analysis failed, results available with viewAerr
    echo analysis failed >> $log
    exit 1
fi
```

B.4 archive

```
#!/bin/ksh

#
#                        Marvel Software Development Environment
#
#                                     Copyright 1989
#                          The Trustees of Columbia University
#                                 in the City of New York
#                                  All Rights Reserved
#
# archive envelope
#
# usage: archive [MODULE]


MT='arch';

attribute='dirname $1'
libdir='dirname $attribute'
arch=${libdir}/'basename $libdir'".a"

mod='basename $1'

echo "$0 $mod on 'date'"
echo

cd $1

ofiles='ls cfiles/*.c/*.o 2>/dev/null'

echo running ar rv on $arch
ar rv $arch $ofiles

if [ $? -eq 0 ]
then
    if [ $MT != "mips" ]
    then
        echo running ranlib on $arch
        ranlib $arch
    fi

    if [ $? -eq 0 ]
    then
```

```
        echo archive now available in ${arch}
        exit 0
    fi
fi

echo archive failed
exit 1
```

B.5 build

```
#!/bin/ksh

#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#
# build envelope
#
# usage: build <PROGRAM>

pgm=`basename $1`
dir=`dirname $1`

echo "$0 $pgm on `date`"
echo

cd $1

echo working directory is `pwd`

# first we need to find all the libraries
# crude way of doing it here

proj_libs=`ls ../../libraries/*/*.a 2>/dev/null`  # this will not be the world
local_libs=`ls libraries/*/*.a *.a 2>/dev/null`   # mods in this program
ofiles=`ls cfiles/*/*.o 2>/dev/null`
ifiles=`ls incs/*/hfiles/* 2>/dev/null`           # hfiles in this program.
                                                  # no support for global incs.

if [ "x$ofiles" = "x" ]
then
    echo "Nothing to build"
    exit 1
fi

idirs=
if [ "x$ifiles" != "x" ]
then
    for f in $ifiles
    do
        idirs="$idirs -I$f"
    done
fi

echo doing the build ...
echo objects are: $ofiles
echo local libraries are: $local_libs
echo project libraries are: $proj_libs

cc -g $idirs $ofiles $local_libs $proj_libs -o $1/$pgm

# this checks for existence, and to be sure it is the proper kind of
# executable.

MT=`arch`

if [ "x$MT" = "xsun4" ]
then
    file $pgm | grep sparc > /dev/null
    ans=$?
elif [ "x$MT" = "xsun3" ]
then
    file $pgm | grep mc680 > /dev/null
    ans=$?
elif [ "x$MT" = "xmips" ]
then
    file $pgm | grep mipsel > /dev/null
    ans=$?
elif [ "x$MT" = "xibmrt" ]
then
    file $pgm | grep executable > /dev/null
    ans=$?
else
    ans=1
fi

if [ $ans -eq 0 ]
then
    echo build successful
    exit 0
else
    echo build failed
    exit 1
fi
```

B.6 buildall

```
#
#                       Marvel Software Development Environment
#
#                                   Copyright 1989
#                         The Trustees of Columbia University
#                                in the City of New York
#                                 All Rights Reserved
#
# buildall envelope
#

echo buildall envelope

BIN=.

echo buildalling executables for project $1 ...
#cc $1/*/*/code.o -o $BIN/prog

if [ $? -eq 0 ]
then
    echo buildall failed
    exit 0
else
    echo executables now available in $BIN/prog
    exit 1
fi
```

B.7 check_in

```
#!/bin/ksh

#
#                      Marvel Software Development Environment
#
#                                  Copyright 1989
#                       The Trustees of Columbia University
#                               in the City of New York
#                                All Rights Reserved
#
# deposit envelope
#

echo deposit envelope
object=`basename $1`

read ans?"deposit $object [y or n]: "

if [ $ans = "y" ]
then
    echo $object deposit
    exit 0
else
    echo $object NOT deposited
    exit 1
fi
```

B.8 check_out

```
#!/bin/ksh

#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# reserve envelope
#

echo reserve envelope
object=`basename $1`

read ans?"reserve $object [y or n]: "

if [ $ans = "y" ]
then
    echo $object reserved
    exit 0
else
    echo $object NOT reserved
    exit 1
fi
```

B.9 compile

```
#!/bin/ksh

#
#                       Marvel Software Development Environment
#
#                                    Copyright 1989
#                         The Trustees of Columbia University
#                                in the City of New York
#                                  All Rights Reserved
#
# compile envelope
#
# usage: compile [CFILE]
#

cd $1
cfile=`basename $1`

echo "$0 $cfile on `date`"
echo

log=c_err

echo "$0 $1 on `date`" > $log
echo >> $log
echo >> $log

# we need to make the -I list

mod_or_prog=`dirname $1`
mod_or_prog=`dirname $mod_or_prog`

ifiles=`ls -d $mod_or_prog/hfiles/* $mod_or_prog/incs/*/hfiles/* 2>/dev/null`

idirs=
if [ "x$ifiles" != "x" ]
then
    for f in $ifiles
    do
        idirs="$idirs -I$f"
    done
fi
```

```
echo "cc -g -c $idirs $cfile" >> $log
cc -g -c $idirs $cfile >> $log 2>&1

if [ $? -eq 0 ]
then
    echo compile successful, results available with viewCerr
    echo compile successful >> $log
    exit 0
else
    echo compile failed, results available with viewCerr
    echo compile failed >> $log
    exit 1
fi
```

B.10 debug

```
#
#                     Marvel Software Development Environment
#
#                                 Copyright 1989
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# debug envelope
#
# usage debug PROGRAM
#

pgm=`basename $1`

echo $0 $pgm

# find the source code

idirs=`ls -d $pgm/* 2>/dev/null`

ilist=

for nextdir in $idirs
do
    if [ -d $nextdir ]
    then
        ilist="${ilist} -I${nextdir}"
    fi
done

dbx $ilist $1/$pgm

if [ $? -eq 0 ]
then
    echo debug successful
    exit 0
else
    echo debug failed
    exit 1
fi
```

B.11 editor

```
#
#                          Marvel Software Development Environment
#
#                                      Copyright 1989
#                            The Trustees of Columbia University
#                                   in the City of New York
#                                     All Rights Reserved
#
# editor envelope
#

file=`basename $1`

echo editor $1 ...
echo

if [ "x$EDITOR" = "x" ]
then
    vi $1/$file
else
    $EDITOR $1/$file
fi

# there is only one exit code here.
exit 0
```

B.12 execute

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# execute envelope
#
# usage execute PROGRAM
#

pgm=`basename $1`

echo execute $pgm ...

$1/$pgm

if [ $? -eq 0 ]
then
    echo execute successful
    exit 0
else
    echo execute failed
    exit 1
fi
```

B.13 list_archive

```
#!/bin/ksh

#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                        The Trustees of Columbia University
#                               in the City of New York
#                                 All Rights Reserved
#
# list_archive envelope
#
# usage: list_archive [LIB]

lib=`basename $1`
dir=`dirname $1`

echo "$0 $lib on `date`"
echo

cd $dir

ar t $lib/${lib}.a

exit 0
```

B.14 release

```
#
#                         Marvel Software Development Environment
#
#                                    Copyright 1989
#                          The Trustees of Columbia University
#                                in the City of New York
#                                  All Rights Reserved
#
# release envelope
#

echo $0 $1

# for Release
#exit 0

# for Maintenance
#exit 1

# for Development
exit 2
```

B.15 viewAerr

```
#
#                       Marvel Software Development Environment
#
#                                    Copyright 1989
#                           The Trustees of Columbia University
#                                in the City of New York
#                                  All Rights Reserved
#
# viewAerr envelope
#

echo viewAerr $1 ...
echo

less $1/l_err

exit 0
```

B.16 viewCerr

```
#                        Marvel Software Development Environment
#
#                                  Copyright 1989
#                         The Trustees of Columbia University
#                               in the City of New York
#                                All Rights Reserved
#
#
# viewCerr envelope
#

echo viewCerr $1 ...
echo

less $1/c_err

exit 0
```

B.17 viewer

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1989
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# viewer envelope
#

echo viewer $1 ...
echo

file=`basename $1`

less $1/$file

# no postconditions to set

exit 0
```

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] Gail E. Kaiser and Peter H. Feiler. Intelligent assistance without artificial intelligence. In *32nd IEEE Computer Society International Conference*, pages 236–241, San Francisco CA, February 1987. IEEE Computer Society Press.