Marvel 3.0
User's Manual

Programming Systems Laboratory
Columbia University
Computer Science Building
500 West 120th St
New York, NY 10027
(212)-854-2736
fax:(212)-666-0140

TR CUCS–033-91

October 25, 1991

The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368, CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

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 (section 6); this manual concentrates on teaching the user how to use the system. The manual consists of the following parts:

- a Tutorial which introduces the user to the MARVEL system and provides a step-by-step guide to creating a sample environment.
- a Reference section that documents each built-in command of MARVEL .
- the C/MARVEL environment which has been used for MARVEL development, complete with rules, envelopes and an objectbase.

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 spend additional 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. Any text that you type in the client's Text Window (figure 2) is "quoted" with double quotes. Any text that you type in the Client Window is in **boldface**. All object names are "quoted", all attribute names are *in this font*, all class names are WRITTEN IN THIS FONT, and all rule names are AS THIS. Anything between angle brackets, < >, is the generic name for that thing which is different from the specific name. For example, the name of any object would be shown as <object name>.

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 and was completed in September 1987. The second large-scale implementation, version 2.6, began in September 1987 and was completed in February 1990.

MARVEL 2.6, was a completely standalone system with a persistent object manager replacing Smile; in no way should this object manager be considered to be a generalized objectbase facility. It was appropriate for exploring the automatic environment generation facilities of MARVEL in a real, usable system. MARVEL 2.6 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.

The basic idea of MARVEL is to automatically generate an environment that supports the specific needs of users 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 which have logical conditions and effects. For example, an editor (rule) can have a condition that the specified module be assigned to the current programmer, and an effect (result of the editor's activity) be that the module's status is not-checked, implying it is necessary to invoke a type checker. The combination of all the rules forms the process model of an environment.
- 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 server supplies the *meta-knowledge* shared by all MARVEL environments, in particular how to construct and maintain the objectbase and how to perform chaining between rules. The project administrator defines the *data* and *process* model which are loaded into the server. The server then tailors its behavior, and provides each of its clients an

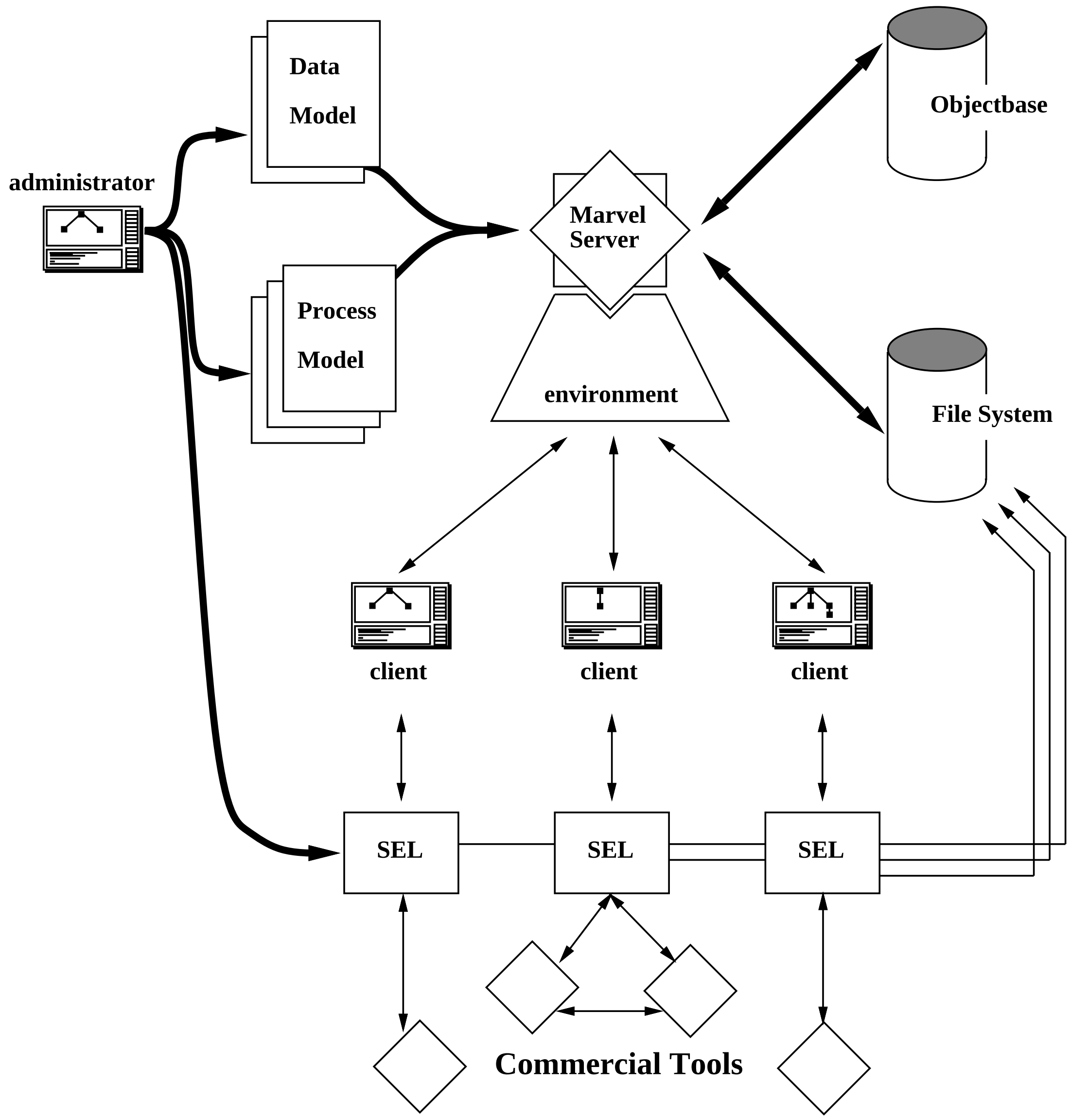

Figure 1: Generating a MARVEL Environment

environment in which to work. The clients can only access information about the objectbase through the server. In addition, when the client is executing an external tool, the tool receives its information through Marvel's Shell Envelope Language (SEL) interface; that is, the information stored on the File System is hidden from the tools. The objectbase can only be accessed by the server.

Marvel has two types of interfaces: a command line interface, and one using X11-windows (version 3 or 4). 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.

2 Tutorial

2.1 Marvel Overview

The long-term goal of the MARVEL project is to develop a kernel for multi-user development environments that allows teams of programmers to cooperate on developing a large-scale software project. The kernel will provide concurrency control and object management primitives that will enable project administrators to build an environment that implements concurrent software process models that describe a spectrum of interactions, ranging from cooperation among members of the same development team to isolation of teams who work on unrelated parts of the project.

MARVEL is a rule-based software development environment kernel that provides assistance in carrying out the software development process. MARVEL is built on top of an object management system that abstracts the components of the project under development as objects and stores them in an objectbase. The software development process of the project is modeled in terms of rules, each of which encapsulates a development activity. MARVEL assists software developers by applying forward and/or backward chaining among the rules, automatically invoking the development activities modeled by these rules.

A project administrator writes a specification of the project data model and process model. Both specifications are written in the MARVEL Strategy Language (MSL). The administrator loads these specifications into the kernel, creating a MARVEL environment that supports both the data management and process management requirements of the project.

The data model is specified in terms of classes, each of which consists of a set of typed attributes that can be inherited from multiple superclasses. In the data model, there is always a unique Top-Level class. There are several varieties of attributes: *small, medium, large* and *link*. Small attributes can be integers, reals, time stamps, user ids, strings, or an enumerated set. Medium attributes are represented as files on the file system. Large attributes are essentially set attributes that contain instances of other classes as their values, thus implementing *composite objects*, and giving the MARVEL object management system (OMS) a hierarchical traversal capability. Set attributes can be restricted in the data model to have only one member; these are called *instance* attributes. Link attributes point to other instances in the objectbase, based upon a specific class, thus giving the MARVEL OMS arbitrary graph traversal capability. Link attributes can also be restricted to have only one link; these are called *instance link* attributes. Existing software systems can be immigrated into MARVEL using the Marvelizer tool. The MARVEL OMS supports creation and deletion of objects according to the data model.

The process model is described in terms of rules that specify the behavior of the tailored MARVEL environment in terms of what commands are available and what kind of assistance is provided. MARVEL supports several models of assistance, ranging from automation to consistency maintenance models. The set of rules that are loaded into a Marvel environment form a network of possible forward and backward chains. Marvel rules are more complicated then their expert systems ancestors; each rule contains a *condition* that must be satisfied for the rule to fire, an activity, which is a general mechanism to execute arbitrary external tools, and multiple *effects* that assert the results of the tool into the Marvel objectbase.

Marvel 3.0 implements a client/server model that supports multiple concurrent users of the same objectbase. Each user requests commands that access the objectbase via the server. Concurrent user commands can cause rule chains to fire concurrently. The server maintains a context for each client and performs the chaining resulting from each client's commands within that client's context. Conflicting accesses by concurrent rule chains to the objectbase are detected by a locking concurrency control protocol that implements granularity locking on the composite object hierarchy. Locks are managed by a lock manager, and there are six basic types of locks, *Shared, Exclusive, Strong Shared, Strong, Exclusive, Intentional Shared, Intentional Exclusive*. These correspond to the lock modes of **S, X, SS, SX, IS, IX**. Locks are internally managed, and the user will never have to manually set locks. This conflict detection protocol provides the bottom layer of a transaction management subsystem. The top layer of the subsystem is a conflict resolution protocol that uses project-specific *control rules* to resolve the conflicts detected by the bottom layer.

2.2 Single-User Tutorial

We now present a single-user tutorial to the C/Marvel environment. In this tutorial, the user will become familiar with several of the Built-In commands provided by the Marvel environment. In addition, the user will gain experience in executing environment-specific rules, and observing forward and backward chaining.

1. Type *make_db* <**marvel_dir**>

   Where <**marvel_dir**> is the name of a directory in which you will be working. Note: <**marvel_dir**> must not already exist.

2. Type *cp_cmarvel* <**marvel_dir**>

   This script will copy the appropriate files into the <**marvel_dir**> directory.

   At this point your environment is ready to run Marvel.

3. Type *marvel_server* <**marvel_dir**>

   This will run the server, with <**marvel_dir**> as the Marvel directory that you set up in step 1.

   If you don't specify the directory, Marvel assumes the current directory, so you could `cd` to <**marvel_dir**> and call the server without arguments.

It is also possible to run the server in the background; in this example, the client has to be run in a different window. To run the server in the background, the user would type "*marvel_server* <**marvel_dir**> &".

Note, when you execute this command the following messages will appear:

```
Error messages will be saved to file Marvel-server.log.
marvel_server: server started on <host>
marvel_server: the objectbase will be saved to file data/objectbase
```

You can verify that the server is running by typing *jobs* (in another window) and the computer will respond with the following:

```
 [NUM] +  Running                    marvel_server <marvel dir>
```

You can run the server in the background, but doing so will prevent you from seeing diagnostic output from the server. It is recommended to run it in the foreground at least the first few times. You can read this information in the Marvel-server.log file created in <**marvel_dir**>.

4. From another window, type

   marvel <**-a**> <**-h hostname**> <**-w**> <**marvel_dir**>

   -a switch tells Marvel to run in administrator mode; default is normal mode. The administrator has additional Built-In commands available that the normal client does not.

   -h hostname specifies the host machine where the server is running; default is current host.

   -w switch tells Marvel to run in the X-window system interface; default is tty (command line) interface.

   <**marvel_dir**> explicitly tells Marvel the working directory – like the server, you can *cd* to <**marvel_dir**> and run the client without specifying the directory; default is current directory

   Note that you can run multiple clients on the same working directory, but only one server can be running for any working directory. Multiple clients are intended for team development. For this example, you should enter:

   marvel -w <**marvel_dir**>

   The following messages are printed to the client's window

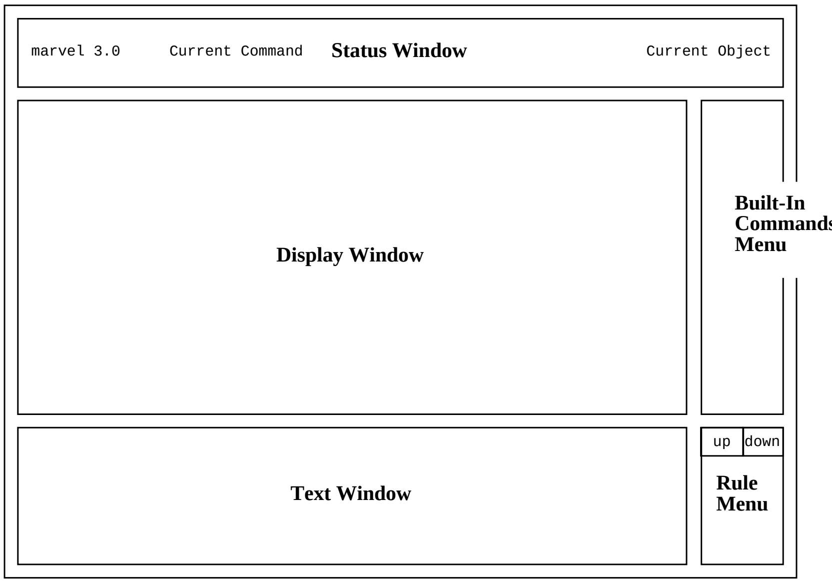

Figure 2: Layout of MARVEL client

```
Marvel Software Development Environment
Columbia University Department of Computer Science
Version 3.0

Error messages will be saved to file Marvel-client.<pid>.log.
Connected to server on <host>
```

An additional message is printed to the Server:

`marvel_server:<user>@<host> logged in.`

It is not necessary to have the server's window continuously visible, but we might refer to it occasionally, so you might prefer to iconify it now.

We are now ready to proceed with the tutorial. Throughout this tutorial, the Client Window refers to the original window in which the *marvel -w* command was entered.

5. We direct your attention to the MARVEL window (figure 2). In the upper left corner, the current version (3.0) is printed. The upper right corner will always describe the "Current Object" in the objectbase. There is always a Current Object (for now, it is "none" since there are *no* objects in the system) and the user can change this focus by clicking on an object with the RIGHT mouse button. The display will automatically register this change.

   There are two panels on the right side of the window: The Built-In Commands Menu and the Rule Menu. The two buttons UP and DOWN in the Rule Menu Panel allow the user to scroll through all the rules loaded into the system. If you click on the DOWN button, for example, the rules that are visible change to *debug* ... *viewErr*. Click on the UP button to restore the first page of the

Rule Menu. For a description of each built-in command, see the descriptions in section 3.

6. We are now ready to begin. We must add a top level instance to create the database. Choose the ADD command, from the Built-In Commands Menu, by clicking on the ADD box with any mouse button. The Built-In Commands Menu is replaced by the Add Menu, which asks the user to select HIER, HORIZ or CANCEL. Note that the command ADD has been displayed in the Status Window. We select horizontal, by clicking on HORIZ. At this point, it makes no sense to add hierarchically since there are no objects in the objectbase to use as a reference. MARVEL is prompting for us to "Enter <instance>", so we type in the name for the new object "Top" and press return. Note in the Display Window there is now a small box labelled "Top". Also observe that the Current Object has automatically changed to this object.

7. Click on this object (to do this, click the LEFT button near the black square labeled top. It doesn't have to be exactly on the square – just near it) to view information about this node. The following is now printed in the lower Text Window:

   Name: Top ID: 0 Owner Class: GROUP
   This is a top level object.

build_status	[(Built, NotBuilt, Initialized)] = Initialized
projects	[set of class PROJECT]

   This tells us the type of object (it belongs to the GROUP class) and that it has two attributes, *build_status* and *projects*. The *projects* attribute states that an object of class GROUP can have children of the class PROJECT in the set attribute *projects*. This is a *large* attribute that defines the structure of the objectbase. The *build_status* attribute is a *small* attribute. These can be integers, reals, timestamps, users, or enumerated sets. In this case, *build_status* is an enumerated set that can take on one of three values "Built", "NotBuilt" or "Initialized" and its current value is "Initialized". In MARVEL these attributes are very important since they are used in rule predicates to enact rule chaining.

8. We note that the Current Object is "Top", so we can add hierarchically (vertically) to populate the objectbase. Adding hierarchically means that the objects will be added to be children of the Current Object. Select the ADD command, and select HIER for adding hierarchically. MARVEL now prompts:

   ```
   Enter <attribute> <name> [<optional subclass name>]
   ```

   Enter in "projects hw1". This tells MARVEL that the name of the object is "hw1" and it is being added to the PROJECT class, since the *projects* attribute is defined to be a set_of PROJECT. We now see two objects on the screen connected vertically by a line that defines a parent/child relationship. Again, click on (or near) the box labelled "hw1" and we will receive the following information:

Name: hw1 ID: 1 Owner Class: PROJECT
Parent Object: Top (class GROUP) Owner Attribute:*projects*

archive_status	[(Archived, NotArchived, Initialied)] = Initialized
build_status	[(Built, NotBuilt, Initialized)] = Initialized
libraries	[set of class LIB]
programs	[set of class PROGRAM]
doc	[set of class DOC]
incs	[set of class INC]
src	[set of class MODULE]

Here, since this object is not a top level instance, MARVEL also prints out information specifying the parent of this object, and the attribute that defines this parent/child relationship. Note that the Current Object has now changed to "Top/projects/hw1". This tells the full path to the object, starting from the top-level instance "Top" through attribute "projects", to the "hw1" object.

9. Now, in similar fashion, we add the following objects:

Object	Action
class: PROGRAM, name "test"	Click on "hw1" with the RIGHT button, and choose the ADD command, HIER, and enter in "programs test"
class: INC, name "includes"	Click on "hw1" with the RIGHT button, and choose the ADD command, HIER, and enter in "incs includes"
class: HFILE, name "main.h"	The Current Object is "Top/projects/hw1/incs/includes" Choose the ADD command, HIER, and enter in "hfiles main.h".
class: MODULE, name "src"	Click on "hw1" with the RIGHT button, and choose the ADD command, HIER, and enter in "src src"
class: CFILE, name "utils.c"	The Current Object is "Top/projects/hw1/src/src". Choose the ADD command, HIER, and enter in "cfiles utils.c"
class: CFILE, name "main.c"	Click on "test" with the RIGHT button, and choose the ADD command, HIER, and enter in "cfiles main.c"

10. Click on "utils.c" with the LEFT button to get information about this object printed out to the MARVEL Text Window. This object has several new attribute types we haven't seen yet: *contents* is a *medium* attribute, that is, a file on the file system which contains the actual C code for this object. There are two types of medium attributes, *text* and *binary*. The *object_code* attribute stores

the resulting object code from this C file. The *reservation_status*, *timestamp*, *owner*, *compile_status*, and *analyze_status* attributes are *small* attributes. The final type of attribute is the *link* attribute. A *link* is a semantic relationship between two objects in the objectbase that falls outside of the standard hiearchical relationships as defined in the composite objectbase. The *ref* attribute is a set of links to other objects from the HFILE class; in this case, this attribute defines the `#include` relationship between C code files and C header files.

Name: utils.c ID: 6 Owner Class: CFILE
Parent Object: src (class MODULE) Owner Attribute:*cfiles*

reservation_status	[(CheckedOut, Available, Initialized)] = Initialized
timestamp	[time] = <Current Date and time>
owner	[user]
compile_status	[(Compiled, NotCompiled, Initialized)] = Initialized
analyze_status	[(Analyzed, NotAnalyzed, Initialized)] = Initialized
compile_log	[text]
analyze_log	[text]
contents	[text]
object_code	[binary]
ref	[set of links of class HFILE]

We now print out the EDIT rule, using the PRINT command in the Built-In Command menu. After selecting PRINT, MARVEL displays the Print Menu. Select RULES, and when MARVEL directs you to click on the rule name, click on the EDIT box found in the Rule Menu (lower right corner). About 40 lines of text are printed; using the arrows, scroll back until the beginning of the newly added text is visible (look for "Click on the rule name." in the Text Window). Figure 3 contains the output.

This is really a lot of information. Let's look at the RULE header first: It specifies the type of object that this edit rule works on (in this case an object from the CFILE class). Note that there are three edit rules in the C/MARVEL system – one each for DOCFILE, HFILE, and CFILE. The main reason why there are three separate ones, is that we want different behaviour for each – this will become apparent later, once you are familiar with chaining. The CONDITION of this rule states that the variable ?c (which is of class CFILE) is bound to an object selected by the user. We will momentarily execute this rule on utils.c, so ?c will be bound as ?c = utils.c. This rule specifies that the object bound to ?c must be owned by the CurrentUser, and that its *reservation_status* attribute must be "CheckedOut".

If we look at the information that MARVEL printed about utils.c, we see that the *owner* attribute is currently undefined, and *reservation_status* is "Initialized". Under these circumstances, the EDIT rule would not be satisfied, so MARVEL would attempt to satisfy the CONDITION (in figure 3) by backward chaining to other rules. We see from the CHAINS output that MARVEL can backward

Marvel Text Window

```
RULE: edit[?c:CFILE]:

CONDITION:
(AND
   ( ?c.owner = CurrentUser )
   ( ?c.reservation_status = CheckedOut))

ACTIVITY:
Tool: EDITOR, operation: edit,
Args: ?c.contents ?c.analyze_status ?c.analyze_log ?c.compile_status ?c.compile_log }

EFFECTS:
0:   (AND
        (?c.analyze_status = NotAnalyzed)
        (?c.compile_status = NotCompiled)
        (?c.timestamp = CurrentTime))
1: NC ( ?c.reservation_status = CheckedOut )

CHAINS:
 backward chains:
(?c.owner = CurrentUser) -->  automation chain to: reserve [?f:FILE ]
(?c.reservation_status = CheckedOut) --> automation chain to:  reserve [?f:FILE]

 forward chains:
(?c.analyze_status = NotAnalyzed) --> automation chain to: dirty[?m:MODULE]
                                                             analyze[?c:CFILE]
(?c.compile_status = NotCompiled) --> automation chain to: dirty[?m:MODULE]
                                                             compile[?c:CFILE]

STRATEGIES:
 edit
```

Figure 3: EDIT rule

chain to the RESERVE rule to make the *reservation_status* = CheckedOut; if we executed the EDIT rule, MARVEL would backward chain to the RESERVE rule. MARVEL would fire the RESERVE rule, which checks out the file (the user has to type in a short description of this file since this is the first time that this file has been reserved), and this would satisfy the CONDITION for EDIT.

MARVEL would then be able to fire the EDIT rule, which it would do, and an `emacs` window would appear on the screen, ready for the user to enter in the file. Once this editing process is done, MARVEL would fire any rules that it can forward chain to. That is, there might be other rules in the system that can now be fired since their CONDITION has been satisfied by the EDIT rule. We see (Figure 3) that MARVEL would forward chain to the DIRTY [MODULE], ANALYZE [CFILE], and COMPILE [CFILE] rules. You should now scroll the text back to the end of the window (i.e., click on the down arrow until the text stops scrolling).

Another way to view this information is through the PRINT GRAPH feature. Select the PRINT rule and when the Print Menu is displayed, select GRAPH. The Display Window now shows a crude representation of the rules and their chaining behavior. Find the box with EDIT(CFILE) inside and click on it with the LEFT button. If you missed and another box has been selected, click anywhere in the Display Window with the RIGHT button to redisplay the rule graph. Note that the EDIT rule has one possible backward chain (the node above labeled RESERVE(FILE), and three forward chains, DIRTY(MODULE), ANALYZE(CFILE) and COMPILE(CFILE). This utility is very useful for determining the chaining

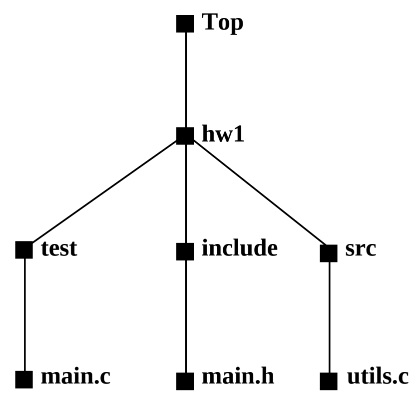

Figure 4: The Current Objectbase

behavior of the given environment. Select DONE to continue the tutorial.

11. We now have 8 objects in the objectbase (see figure 4). However, the C files and the H files are empty. To actually write code, we can't use any of the built-in commands, since they are used only to build the structure of the objectbase. We must use one of the user-provided rules.

    In the Rule Menu (lower right corner), we see a box labeled EDIT. This corresponds to a rule that edits a file. What we will now do is edit the two C files and add code to them. Select the EDIT rule, and when MARVEL prompts for the rule arguments, click on the "utils.c" object (or type "utils.c") and press return. We must keep an eye on the window with the client in it, since we will soon be asked to enter in the description of "utils.c" when MARVEL backward chains to the RESERVE rule. As an example (see figure 5), we enter **contains useful functions**, press return, and then enter a line containing only a "." Once the RESERVE rule has finished processing, the EDIT rule goes to work, placing on the screen an `emacs` window. For this tutorial, you should enter in the following C file:

```
#include <stdio.h>

void sub1()
{
  printf("sub1: Beginning\n");
  printf("sub1: End\n");
}
```

    We now leave `emacs`, (CTRL-X CTRL-C), thus completing the EDIT rule. MARVEL will now execute all the forward chains it can, resulting in figure 5.

12. Click on "utils.c" with the LEFT button, and note that both the *reservation_status* has changed to "CheckedOut" and the *owner* attribute is defined as "<your user-id>". In addition, the *analyze_status* attribute is set to "Analyzed" and the *compile_status* attribute is set to "Compiled."

Client Window

```
Now reserving .../Top/projects/hw1/src/src/cfiles/utils.c/utils.c
Now creating RCS directory...
.../Top/projects/hw1/src/src/cfiles/utils.c/RCS/utils.c,v <--
    .../Top/projects/hw1/src/src/cfiles/utils.c/utils./c
initial revision: 1.1
enter description, terminated with single '.' or end of file:
NOTE:  This is NOT the log message!

contains useful functions
.

>> >> done
.../Top/projects/hw1/src/src/cfiles/utils.c/utils.c reserved

Editing utils.c
Changes Made and saved.
analyzing utils.c on Mon Sep 16 18:35:51 EDT 1991

lint utils.c
analysis successful, results available with viewErr
compiling utils.c on Mon Sep 16 18:35:56 EDT 1991

compile successful, results available with viewErr
```

Marvel Text Window

```
FIRING RULE : edit (utils.c)
  PRECONDITIONS for edit( utils.c)
  Can backward chain to ...
   reserve( utils.c )

  EXECUTING: reserve( utils.c )

  EXECUTING: reserve( utils.c )
  EXECUTION RESULT : reserve( utils.c ) has satisfied precondition of edit (utils.c )
  RE-EVALUATING: edit(utils.c)

  EXECUTING : edit( utils.c )

  EXECUTING : edit( utils.c )
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        edit(utils.c)
  Can forward chain to ...
    analyze(utils.c)

  EXECUTING analyze( utils.c )
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        analyze(utils.c )
  Can forward chain to ...
    compile(utils.c )

  EXECUTING compile( utils.c )
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        compile(utils.c )
Execution cycle completed
```

Figure 5: Result of EDIT[utils.c]

13. Moving on, we now add code to the "main.c" object. Click on the EDIT rule, and then click on the "main.c" object with the LEFT button, and press return. Once again, MARVEL will have to backward chain to the RESERVE rule, so you should have the Client Window available to type in a description of this file. In figure 5 we enter in **contains the main procedure**, followed by a line with a ".". When the `emacs` window appears, enter in the following C file.

```
#include <stdio.h>
#include "main.h"

main()
{
  printf("Version: %s\n", VERSION);
  printf("This is the main file.\n");

  sub1();
}
```

Now save this file and leave `emacs`, (CTRL-X CTRL-C), and watch MARVEL execute its chaining cycle (figure 6). What happened? Well, click on the "main.c" object with the LEFT button, to print out its information.

Name: main.c ID: 7 Owner Class: CFILE
Parent Object: test (class PROGRAM) Owner Attribute: *cfiles*
reservation_status [(CheckedOut, Available, Initialized)] = CheckedOut

Client Window	**Marvel Text Window**
Now reserving .../Top/projects/hw1/programs/test/cfiles/main.c/main.c Now creating RCS directory... .../Top/projects/hw1/programs/test/cfiles/main.c/RCS/main.c,v <-- .../Top/projects/hw1/programs/test/cfiles/main.c/main.c initial revision: 1.1 enter description, terminated with single '.' or end of file: NOTE: This is NOT the log message! **`contains the main procedure`** . > >> done .../Top/projects/hw1/programs/test/cfiles/main.c/main.c reserved Editing main.c. Changes Made and saved. analyzing main.c on Mon Sep 16 19:28:08 EDT 1991 > lint main.c analysis failed, results available with viewErr	FIRING RULE : edit (main.c) PRECONDITIONS for edit(main.c) ARE NOT SATISFIED Can backward chain to ... reserve (main.c) EXECUTING : reserve (main.c) EXECUTING : reserve (main.c) EXECUTION RESULT : reserve (main.c) has satisfied precondition of: edit(main.c) RE-EVALUATING : edit (main.c) EXECUTING : edit (main.c) EXECUTING : edit (main.c) EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING: edit(main.c) Can forward chain to ... analyze(main.c) EXECUTING analyze(main.c) EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING: analyze(main.c) Execution Cycle completed

Figure 6: Result of EDIT[main.c]

timestamp	[time] = <Current Date and time>
owner	[user] = <Current User>
compile_status	[(Compiled, NotCompiled, Initialized)] = NotCompiled
analyze_status	[(Analyzed, NotAnalyzed, Initialized)] = NotAnalyzed
compile_log	[text]
analyze_log	[text]
contents	[text]
object_code	[binary]
ref	[set of links of class HFILE]

It appears that the file was successfully reserved, but there were problems during the analysis of "main.c". Let's EDIT the "main.c" file to see what the problem was. Select the EDIT rule, and then with the LEFT button, click on "main.c" and press return.

When the `emacs` window appears, there are several buffers besides "main.c". The middle buffer, "main.canalyze_log", contains the results of what happened when lint was run on "main.c"; the lower buffer is currently empty since it contains the results of what happened when `cc` is run on main.c, and that hasn't been done yet. If we look at the "main.canalyze_log" buffer we see that the analysis failed because lint was unable to find the "main.h" file. This is rather obvious, since we haven't created this file yet. Leave `emacs` now without making any changes to "main.c". MARVEL takes no action, since the object has not been changed.

14. Select the EDIT rule, and click with the LEFT button on the "main.h" object, and press return. MARVEL will once again backward chain to RESERVE, so be prepared to enter in a short description for this file. We enter **main header file**, followed by a line with a "." and when the `emacs` window appears, enter in the following short header file:

```
#ifndef _MAIN_H
#define _MAIN_H

#define VERSION "1.0"

#endif
```

Now leave `emacs`.

15. We now manually analyze "main.c" by selecting the ANALYZE rule (We may have to click on the UP in the Rule Menu to make the ANALYZE rule visible), clicking on the "main.c" object with the LEFT button, and pressing return (figure 7). Well, what happened? Basically, Marvel doesn't know that the HFILE main.h is a C header file, and that the `lint` tool needs it. We can see this information readily by looking at the ANALYZE rule. Select the PRINT command from the Built-In Command Menu. When the Print Menu appears, select RULES, and then click on the ANALYZE rule in the Rule Menu (again, we may have to make the ANALYZE rule visible by clicking on the UP button). The output is in figure 10.

Client Window	**Marvel Text Window**
analyzing main.c on Mon Sep 16 19:49:57 EDT 1991	FIRING RULE : analyze (main.c)
lint main.c analysis failed, results available with viewErr	EXECUTING : analyze(main.c) EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING: analyze(main.c)
	Execution Cycle completed

Figure 7: Result of ANALYZE[main.c]

The CONDITION part has a derived parameter ?h, which is bound at runtime to all those HFILEs that ?c links to through the *ref* attribute. Since ?c (main.c) currently has no links set, the ANALYZE rule does not see the HFILEs. This is especially important since the ACTIVITY section sends to the shell script (which invokes the `lint` tool) the header file contents. We realize that we have not set things up correctly, so we make the link right now. If the Text Window is not scrolled down to the end, we do that now by clicking on the down arrow until the text scrolls to the end.

16. Select the LINK command from the Built-In Command Menu, and Marvel will prompt "Pick the source object or pick done". Click with the LEFT button on "main.c" and Marvel will ask for the link attribute. Since there is only one link attribute for the CFILE class, Marvel automatically chooses the *ref* attribute and replies "Pick the source link attribute: ref". Now Marvel is prompting for the destination object, so click on "main.h" with the LEFT button.

    Note that Marvel has drawn a curved, undirected arc between these two objects. When the objectbase is displayed, all links are curved, and all parent/child relationships are straight lines. Click on "main.c" with the LEFT button to see how the new information is displayed. Note that now the *ref* attribute is defined to point to "main.h" (with ID of 4 to disambiguate.) Back links are also printed, which can be checked by clicking the LEFT button on "main.h" and noticing that the third line of output says "Linked to by object main.c (class CFILE)".

17. Now let's manually analyze "main.c" once again by selecting the ANALYZE rule and clicking on "main.c" and pressing return. Note that the ANALYZE rule has completed successfully, and that it forward chains to the COMPILE rule. This rule also completes successfully, and MARVEL chains to the BUILD rule, which fails (figure 11). To find out why this BUILD rule failed, select the VIEWERR rule defined in the Rule Menu (again, it might not be visible, so use the UP/DOWN buttons to bring it into view). Select it, and click on "test" with the LEFT button, and press return (figure 8).

Client Window

```
======================= build errors ========================
build .../Top/projects/hw1/programs/test/testexec on
   Mon Sep 16 23:25:12 EDT 1991
ld: Undefined symbol
  _sub1
```

Figure 8: Result of VIEWERR[test]

18. For some reason, the BUILD rule was unable to find function `sub1` from "utils.c". So now we look at the BUILD rule for PROGRAM. When we print out the BUILD rule, use the scrollbar to find the BUILD rule for class PROGRAM (see figure 10). So we see that we have left something out; we need to add libraries to this project since the BUILD rule requires all the CFILEs not owned by the specific PROGRAM to be archived into an archived library file (.a file). If the Text Window is not scrolled down to the end, we do that now by clicking on the down arrow until the text scrolls to the end.

19. We need to add an object of class LIB to be a child of hw1. From the BUILD rule we see that we need to add it to the *libraries* attribute of "hw1". So, we change our Current Object to "hw1" by clicking on "hw1" with the RIGHT button. Note how the Current Object changes to "Top/projects/hw1". Next, select the ADD command, specify HIER, and enter the attribute name followed by the object name "libraries lib". Now we want to tell MARVEL that the CFILE "utils.c" should be archived into this library. There is no attribute in the CFILE class object to specify this, but we once again see from the BUILD rule that objects from the MODULE class contain a link to a LIB class object. If we click on "src" object with the LEFT button, we will see the following:

    Name: src ID: 5 Owner Class: MODULE
    Parent Object: hw1 (class PROJECT) Owner Attribute: *src*
    archive_status [(Archived, NotArchived, Initialized)] = Initialized

modules	[set of class MODULE]
cfiles	[set of class CFILE]
library	[set of links of class LIB]

So the data model is configured to have each module point to a library in which it will archive its CFILE objects. We set the link now between "src" and "lib" in a similar fashion as before. Select the LINK command, click on "src" with the LEFT button when MARVEL prompts with "Pick the source object or pick done". MARVEL automatically chooses the *library* attibute ("Pick the source link attribute: library") since there is only one possible link attribute to choose from. Click on the "lib" object with the LEFT button to fix the destination of this link when MARVEL prompts with "Pick the destination object or pick done". A curved arc will appear between "src" and "lib".

20. Now that we have set this link, we must archive this module, so select the ARCH rule from the Rule Menu and click on "src", pressing return when done (see figure 9). Note that MARVEL forward chains to the rule which archives a library, and then continues to automatically build the "test" program.

Client Window

```
running ar rv on lib.a.

ar: creating .../Top/projects/hw1/libraries/lib/lib.a
a - .../Top/projects/hw1/src/src/cfiles/utils.c/utils.c.o
running ranlib on
archive now available in lib.a
build .../Top/projects/hw1/programs/test/testexec on
Mon Sep 16 23:27:19 EDT 1991
cc .../Top/projects/hw1/programs/test/cfiles/main.c/
main.c.o .../Top/projects/hw1/libraries/lib/lib.a -ll -lc
-lm -o .../Top/projects/hw1/programs/test/testexec
build successful
```

Marvel Text Window

```
FIRING RULE: arch(src)

 EXECUTING: arch(src)
 EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                  arch(src)

 EXECUTING: arch(lib)
 EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                  arch(lib)
 Can forward chain to ... build(test)

 EXECUTING : build(test)
 EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                build(test)
 Can forward chain to ... build (hw1)
 EXECUTING: build(hw1)
 EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                 build(hw1)
Execution Cycle completed
```

Figure 9: Result of ARCH[src]

21. We are now ready to run the program! Choose the EXEC_PROG rule from the Rule Menu, and click on "test" with the LEFT button, pressing return when done. Find the client window; it is waiting for a list of parameters to give to "test". Press return since there are none, and the program executes (figure 12).

Marvel Text Window

```
RULE: analyze [?c:CFILE]

CONDITION:
(forall HFILE ?h suchthat   (LINKTO [?c.ref ?h]))
:
NB (?c.analyze_status = NotAnalyzed)

ACTIVITY:
Tool: COMPILER, Operation: analyze,
Args: ?c.contents ?c.analyze_log ?h.contents

EFFECTS:
0: (?c.analyze_status = Analyzed)
1: [?c.analyze_status = NotAnalyzed]

CHAINS:
forward chains:
(?c.analyze_status = Analyzed) --> automation chain to:
                                          compile[?c:CFILE]
forward chains:
[?c.analyze_status = NotAnalyzed] -->consistency chain to:
                                          dirty[?m:MODULE]
STRATEGIES:
 compile
```

```
RULE: build [?p:PROGRAM]

CONDITION:
   (AND (forall PROJECT ?P   suchthat (MEMBER [?P.programs ?p]))
            (forall MODULE ?m  suchthat (MEMBER [?P.src ?m])
            (forall LIB ?l             suchthat (LINKTO [?m.library ?l]))
            (forall CFILE ?c        suchthat (MEMBER [?p.cfiles ?c])))
:
(AND (?c.compile_status = Compiled)
         (?l.archive_status = Archived))

ACTIVITY:
Tool: BUILD, Operation: build_program,
Args: ?c.object_code ?p.exec  ?p.build_log  ?l.afile

EFFECTS:
0: NF(?p.build_status = Built)
1: NF(?p.build_status = NotBuilt)

CHAINS:
backward chains:
(?c.compile_status = Compiled) --> automation chain to:
                                          compile[?c:CFILE]
backward chains
(?l.archive_status = Archived) --> automation chain to:
                                          arch[?l:LIB]
STRATEGIES:
  build
```

Figure 10: The ANALYZE and BUILD rules

Client Window

```
lint main.c
analysis successful, results available with viewErr
compiling main.c on Mon Sep 16 23:25:06 EDT 1991

compile successful, results available with viewCerr
build .../Top/projects/hw1/programs/test/testexec on
   Mon Sep 16 23:25:11 EDT 1991
cc  .../Top/projects/hw1/programs/test/cfiles/main.c/main.c.
   -ll -lc -lm -o .../Top/projects/hw1/programs/test/testexec
build failed
```

Marvel Text Window

```
FIRING RULE : analyze(main.c)
  EXECUTING: analyze(main.c)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        analyze(main.c)
  Can forward chain to ...compile (main.c)

  EXECUTING: compile(main.c)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        compile(main.c)
  Can forward chain to ... build(test)

  EXECUTING: build(test)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        build(test)
 Can forward chain to ... dirty (hw1)

  EXECUTING: dirty(hw1)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        dirty(hw1)
  Can forward chain to ... clean (main.h)
  EXECUTING: clean (main.h)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        clean (main.h)
Execution Cycle completed
```

Figure 11: Correct Result of ANALYZE[main.c]

Client Window

```
executing .../Top/projects/hw1/programs/test/testexec on
    Mon Sep 16 23:58:45 EDT 1991

Enter in the parameters for
    .../Top/projects/hw1/programs/test/testexec:

<No Parameters, press return>

Version: 1.0
This is the main file.
sub1: Beginning
sub1: End
```

Marvel Text Window

```
FIRING RULE: exec_prog(test)
  EXECUTING: exec_prog(test)
 EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        exec_prog(test)

Execution Cycle completed
```

Figure 12: Result of EXEC_PROG[test]

22. To finish the tutorial, execute the DEPOSIT rule on the three files we have edited – "main.c", "main.h" and "utils.c". In each case, the Client Window will prompt the user to verify the deposit of the file (enter "y") and then ask for a message (we type in "done"). This completes the single-user tutorial.

23. When you are done with MARVEL, select the QUIT command from the Command Menu. When the MARVEL Window has disappeared, go to the the Client Window, and enter *kill_server*. This will kill the server process. Alternatively, you could go to the window with the marvel_server running, and type CTRL-C to stop the program.

24. THAT'S IT! You should now take a close look at the strategy files (with .load suffix) that are in <**marvel_dir**> to become familiar with the rules in each of them. You should also look at the shell scripts (no suffix). Each shell script (e.g., build) is written in MARVEL's SEL, Shell Envelope Language, and has been compiled into an appropriate KSH/SH/CSH shell file (with .env suffix). You should be familiar with these as well.

25. Please note: For now don't change the Data Model as defined in data_model.load since it will invalidate the objectbase that you have created, and you will have to start from scratch. There are internal ways to update the objectbase (e.g., to add/subtract an attribute to/from a certain class), but we haven't explained them here. Rules can be added/deleted but you should be well acquainted with C/MARVEL before attempting this. Good Luck!

3 Marvel

3.1 Built-In Commands

This section describes each of the built-in commands available to the user. These commands are available in all MARVEL environments, and are located in the Built-In Command menu, the upper right panel of the MARVEL client window. Environment-specific commands match rules as defined by the project administrator; for example, a command COMPILE that matches a rule designed to compile a C file. Built-in commands directly manipulate the structure of the objectbase. Each built-in command is described in detail, and a list of common error cases is provided.

3.1.1 add

The ADD command adds a new object the objectbase.

MARVEL maintains a concept of a "Current Object". Whenever an object is added, it's location in the objectbase is determined relative to the Current Object. The Current Object can be changed with the CHANGE command.

There are two ways to add an object: horizontally and hierarchically (vertically). A Marvel objectbase is a directed graph, which conforms to a structure as defined in the *data model*. The objectbase is constructed by composite objects. That is, an object has an attribute (called a set attribute) which is itself a collection of objects from a specific class. Adding an object horizontally makes the new object a sibling to the Current Object. That is, the new object will have the same parent that the Current Object has, and belong to the same set attribute. Adding an object hierarchically makes the new object a child of the Current Object, belonging to the user-specified set attribute (figure /refadd-command). After adding an object, the screen will refresh to show the new display.

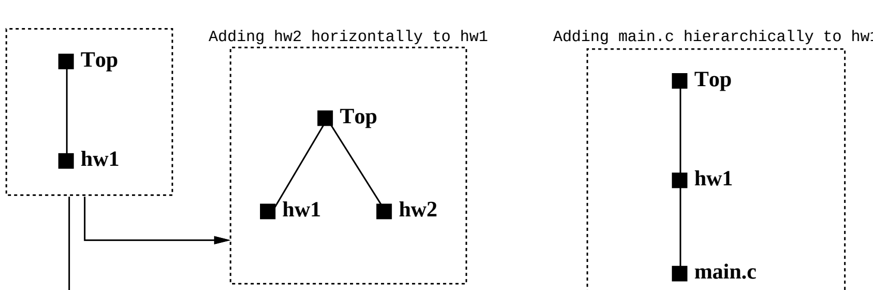

Figure 13: Adding an object

If an optional sub-class name is entered by the user, then the object to be added will be classified as an object of that sub-class, instead of the class of the set attribute. This operation will only be allowed if the sub-class name provided is actually a sub-class of the class of the set attribute. When the client selects the ADD command, the Add Menu is displayed, prompting the user for HIER, HORIZ, or CANCEL.

1. Click on ADD
2. Click on HORIZ, HIER or CANCEL
3. (HIER) Type in the attribute and the object name
   (HORIZ) Type in the object name.
4. Type in the <optional sub-class name>
5. Press return.
6. Confirm the addition by clicking YES, or cancel the addition by clicking NO.

Errors:

If the optional sub-class is not a valid class, or not a valid sub-class for the given set attribute, then the ADD command is aborted.

If the attribute is not valid for the Current Object, MARVEL will reply "Template attribute <attribute> not found. ADD failed".

If the user is attempting to add the first instance hierarchically, MARVEL responds "The first instance can not be added downward. add failed."

If the Current Object already has an object in the user-specified set attribute with the given name, MARVEL will reply "There already is an object with name <object_name> in set attribute <attribute>."

If the attribute chosen by the user is not a set attribute (i.e., it is an instance attribute which can have at most one member), and the user attempts to add more than object to this attribute, MARVEL responds with "<Current Object>.<attribute> is already assigned(<the object>). ADD failed".

Concurrency Errors:

If another client is currently accessing the Current Object in write (exclusive) mode, then the ADD command will be aborted, and the following message will be printed:

> Concurrent access conflict on object <object_name>
> Your command requested accessing <object_name> in write (exclusive) mode
> But user <other client> is running the <rule> rule, which is accessing <object_name> in write (exclusive) mode
> Aborting the transaction. add: tx_WRITE on Current Object failed

If another client has deleted the Current Object, and the current command is to add an object hierarchically, MARVEL will reply "Current Object is invalid. Refreshing your screen. Add failed." This occurs because the display has become out of date with respect to the objectbase.

3.1.2 browse

The BROWSE command allows the user to graphically navigate through the objectbase, and selectively view any desired sub-tree.

The objectbase displayed to the user can be in one of two states: *Top-Level*, or *Sub-Tree*. Since there can be multiple objects belonging to the Top-Level class, the objectbase is actually a forest of object trees. The only way to view all of the trees at once is to be in the Top-Level state. When the objectbase is being displayed in the Sub-Tree state, the object that is the root of the sub-tree is centered in the top row of the screen. Note: The Current Object focus is unchanged by any user browsing.

When the user selects the BROWSE commands, the *browse menu* replaces the Command Menu, and the client is in Browse Mode. The user can choose from *zoomin, zoomout, pan, info* or *done*.

1. *zoomin* allows the user to choose an object on the screen to be the next sub-tree root.
2. *zoomout* expands the sub-tree up one level, and makes the parent of the current sub-tree root, the new sub-tree root. This is only applicable on the current object, which is the root of the given sub-tree.
3. *pan* allows the user to move horizontally in the tree (left or right) from a given object. This command is a shorthand for zoomout followed by zoomin.
4. *info* prints out the information for the selected node.
5. *done* leaves the Browse mode.

Figure 14 shows the various options available to the user. Note that while the Browse mode is active, the three mouse buttons change their functionality:

- *Left* pans left to the next sub-tree. If there is no sub-tree to pan left to, MARVEL responds with "Cannot pan left. This is a leftmost node."
- *Middle* is an accelerator for *zoomin/zoomout*. When the user clicks on any object except the current sub-tree root object, that object becomes the new sub-tree root (*zoomin*). When clicked on the current sub-tree root, MARVEL expands up the sub-tree one level, and makes the parent of the current sub-tree root, the new sub-tree root (*zoomout*).
- *Right* pans right to the next sub-tree. If there is no sub-tree to pan right to, MARVEL responds with "Cannot pan right. This is a rightmost node."

Errors:

None. However, it is possible that the objectbase being browsed is out of date with the current objectbase, since other clients might have deleted or added objects; this will be transparent to the client. The only client interaction that might be impaired is the *info* option to print information about a given object. If the object chosen by the user has been deleted by some other client, then MARVEL will not be able to print any information for that object (obviously!).

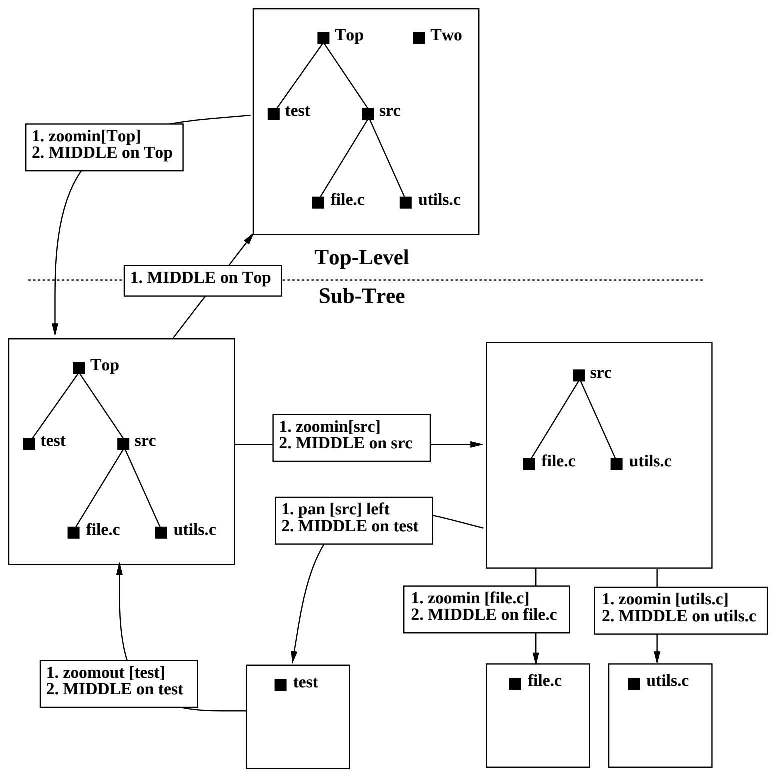

Figure 14: Browsing capabilities

3.1.3 change

The CHANGE command allows the user to change the Current Object focus, as displayed in the upper right corner of the screen. When the user selects the CHANGE command, MARVEL prompts the user to "Pick an object to change to", and waits until the user has clicked on an object. The Current Object focus is immediately changed to this object. The RIGHT button is an accelerator for this command – click on any object with the RIGHT button, and it will become the Current Object.

Concurrency Errors:

If the user selects an object which another client has deleted, MARVEL responds "Object not found. Refreshing screen" and the display is refreshed.

3.1.4 copy

The COPY command allows the user to copy objects in the objectbase to be children of other objects, as allowed by the *data model*.

It is not possible to change the class of a given object, once the object has been created and given a specific type. However, an object can be copied to an attribute whose owner class is a superclass of the object's class. An object cannot be copied to itself (i.e., if the set attribute is that same as the attribute that the Source Object belongs to, the Target Parent Object cannot be the parent of Source Object).

When an object is copied, the entire sub-tree rooted at that object is copied as well. All small and medium attributes are also copied, but *link* attributes are not.

Top-Level objects, as defined in the *data model*, cannot be copied. In addition, objects cannot be copied to be children of themselves, or children of one of their descendents. This would violate the hierarchy of the objectbase.

1. Click on COPY
2. Pick the Source Object
3. Pick the Target Parent Object
4. (If more than one large attribute in Target Parent Object) Pick the target attribute, otherwise the only attribute is automatically chosen.
5. Confirm the copy of <Source Object> to <Target Parent Object>.<attribute> by clicking YES, or cancel the copy by clicking NO.

Errors:

If the Source Object belongs to a class that is incompatible with the set attribute (i.e., not the same class, and not a sub-class of the set attribute class), then MARVEL responds with "Copy error: Class <Source Object Class> of object <Source Object> is not compatible with attribute <attribute> of class <attribute class>", and the COPY command is aborted.

If the Target Parent Object currently has an object in the given set attribute with the same name as the source object, MARVEL reponds with "Copy error: Object already exists in <attribute> with name <Source Object Name>", and the COPY command is aborted.

If the Target Parent Object is the Source Object, or a descendent of the Source Object, MARVEL responds with "Copy error: Can't copy an object to be a descendent of itself", and the COPY command is aborted.

Concurrency Errors:

If another client is currently accessing either the Target Parent Object or the Source Object in write (exclusive) mode, then the COPY command will be aborted, and the following message will be printed:

> Concurrent access conflict on object <object_name>
> Your command requested accessing <object_name> in write (exclusive) mode
> But user <other client> is running the <rule> rule, which is accessing <object_name> in write (exclusive) mode
> Aborting the transaction. copy_cmd: tx_WRITE on Current Object failed

3.1.5 delete

The DELETE command allows the user to delete objects from the objectbase.

Since the *data model* is constructed with composite objects, deleting an object deletes the entire sub-tree rooted at that object. However, any objects linked to from within this sub-tree are *not* deleted. Once the object has been deleted, the new objectbase image is displayed.

1. Click on DELETE
2. Pick an object to delete
3. Confirm the deletion of <Object> by clicking YES, or cancel the deletion by clicking NO.

Errors:

If the client clicks too low in the MARVEL window, MARVEL responds with "Delete: Invalid Pick", and the DELETE command is halted. If this happens, the client must select the DELETE command again.

Concurrency Errors:

If another client is currently accessing the chosen object in write (exclusive) mode, or accessing an object within the sub-tree in write (exclusive) mode (thus accessing the chosen object in intentional write mode), then the DELETE command will be aborted, and the following message will be printed:

> Concurrent access conflict on object <object_name>
>
> Your command requested accessing <object_name> in strong write (no children can be accessed) mode But user <other client> is running the <rule> rule, which is accessing <object_name> in write (exclusive) mode
> Aborting the transaction

If another client has deleted the chosen object, then MARVEL will respond "Object not found, refreshing your screen...", and the objectbase image will be refreshed.

3.1.6 execute

The EXECUTE command allows the user to execute an arbitrary MARVEL command script. A script is a file with a special header of 15 characters: `#!marvel script`. Within the file is a list of MARVEL commands to execute. Since they will be executed without graphical interaction, the commands are written using the *Command Line* interface, and any user input will be queried as determined by this interface. For full specifications of this interface, we refer the reader to section 3.2. This command essentially provides a batch interface to MARVEL.

3.1.7 help

The HELP command allows the user to get more information about a variety of subjects.

When the client selects the HELP command, the Help Menu is displayed with the following options:

- *?* outputs the current listing of help available, broken down into the three sections: *Subject, Command* and *Rule*.
- *Command* prompts the client to select a command from the Command Menu. Any information about the command is printed to the client's Text Window in a fashion similar to the UNIX command `man`.
- *Subject* prompts the client to enter in one of the topics as printed in the *Subject* section from above, and information about this subject is printed to the client's Text Window.
- *Rule* prompts the client to select a rule from the Rule Menu. Any information about the rule is printed to the Output Window in a fashion similar to the UNIX command `man`.

3.1.8 link

The LINK command allows the user to link an object to another object in the objectbase, as allowed by the *data model*.

A *link* is a semantic relation between two objects, as defined in the *data model*, that does not conform to the strict hierarchical composition of the objectbase. The link is rooted at one object (the source) and links to another object (the destination). The destination instance and the source instance may be the same object.

1. Click on LINK
2. Pick the source object or DONE
3. Pick the source link attribute (if there is only one link attribute, it is chosen automatically) from the provided menu
4. Pick the destination object

Errors:

If the source object selected by the client has no link attributes, MARVEL responds with "link aborted: selected object has no link attributes."

If the destination object is not the same class as, or not a sub-class of, the link attribute class, MARVEL responds with "Incompatible Types."

If the destination object is already being linked to through this link attribute, MARVEL responds "object <destination object> already linked by this attribute."

If the link attribute is a single link, and the user is attempting to create another link, MARVEL responds "single link attribute already assigned."

Concurrency Errors:

If another client has created the link from <source object> to <destination object>, MARVEL responds "object <destination object> already linked by this attribute", and the screen will refresh to show the link.

If another client has deleted either the <source object> or the <destination object>, MARVEL responds "Object Not Found. Refreshing your screen." and the most recent objectbase will be displayed.

3.1.9 move

The MOVE command allows the user to move objects in the objectbase to be children of other objects, as allowed by the *data model*.

It is not possible to change the class of a given object, once the object has been created and given a specific type. However, an object can be moved to an attribute whose owner class is a superclass of the object's class. An object cannot be moved to itself (i.e., the Target Parent Object cannot be the parent of Source Object).

When an object is moved, the entire sub-tree rooted at that object moved, as well. Top-Level objects, as defined in the *data model*, cannot be moved. In addition, objects can not be moved to be children of themselves, or children of one of their descendents. This would violate the hierarchy of the objectbase.

1. Click on MOVE
2. Pick the Source Object
3. Pick the Target Parent Object
4. (More than one large attribute in Target Parent Object) Pick the target attribute, otherwise the only attribute is automatically chosen.
5. Confirm the move of <Source Object> to <Target Parent Object>.<attribute> by clicking YES, or cancel the move by clicking NO.

Errors:

If the Source Object belongs to a class that is incompatible with the set attribute (i.e., not the same class, and not a sub-class of the class), then MARVEL responds with "Move error: Class <Source Object Class> of object <Source Object> is not compatible with attribute <attribute> of class <attribute class>, and the MOVE command is aborted.

If the Target Parent Object currently has an object in the given set attribute with the same name as the source object, MARVEL reponds with "Move error: Object already exists in <attribute name> with name <Source Object Name>", and the MOVE command is aborted.

If the Target Parent Object is the Source Object, or a descendent of the Source Object, MARVEL responds with "Move error: Can't move an object to be a descendent of itself", and the MOVE command is aborted.

If the user tries to move an object to the same set attribute that it currently belongs to, MARVEL replies with "You can't move an object to itself."

Concurrency Errors:

If another client is currently accessing either the Target Parent Object or the Source Object in write (exclusive) mode, then the MOVE command will be aborted, and the following message will be printed:

> Concurrent access conflict on object <object_name>
> Your command requested accessing <object_name> in write (exclusive) mode
> But user <other client> is running the <rule> rule, which is accessing <object_name> in write (exclusive) mode
> Aborting the transaction. move_cmd: tx_WRITE on Current Object failed

3.1.10 print

The PRINT command allows the user to display the definitions of various components of the environment. The LEFT button is an accelerator for PRINT INST. When the user selects the PRINT command, the Print Menu replaces the Command Menu, and the user can choose from the following options:

- *class* prints out the definitions of all the loaded classes
- *inst* prints out information for an object that the user chooses
- *rules* prints out the definition of a rule that the user chooses
- *current* prints out information about the current object
- *Graph* displays a graph of all the rule chaining connections within the given rule set. Consistency implications are represented by dashed lines, while automation implications are drawn as solid lines. The default view shows every rule in the system. To focus on any one rule, click with the LEFT button on the box on the screen that contains that rule. To return to the default view, click on any rule with the RIGHT button. When focusing on a rule, backward chains are represented as lines originating at the center node (rule) rising vertically to nodes drawn above the original rule. Forward chains are represented as lines originating at the center node (rule) falling vertically to nodes drawn below the original rule.
- *Done* leaves the Print Mode.

Concurrency Errors:

If another client has deleted the <object> for which the user requests to view information, MARVEL responds "Object Not Found. Refreshing your screen." and the most recent objectbase will be displayed.

3.1.11 quit

The QUIT command terminates the client process.

3.1.12 refresh

The REFRESH command redisplays the client's objectbase. This is necessary when the objectbase has changed, and the client's objectbase image is out of date.

3.1.13 rename

The RENAME command allows the user to get more information about a variety of subjects.

1. Click on RENAME
2. Pick the object to rename, or pick DONE
3. (if not DONE) Enter new name for <object>

Errors:

If the set attribute that contains <object> already has a member with the new name entered by the client, MARVEL responds with "<new name> must be unique amongst the children of <parent object>. rename of <object> failed."

Concurrency Errors:

If another client is currently accessing either the Target Parent Object or the Source Object in write (exclusive) mode, then the RENAME command will be aborted, and the following message will be printed:

> Concurrent access conflict on object <object_name>
> Your command requested accessing <object_name> in write (exclusive) mode
> But user <other client> is running the <rule> rule, which is accessing <object_name> in write (exclusive) mode
> Aborting the transaction. rename_cmd: tx_WRITE on Current Object failed

3.1.14 set

The SET command allows the user to modify several parameters of the run-time environment. When this command is selected, the Set Menu is displayed with the following options:

Show all	Output current values
auto_update_display	*** Unused ***
verbose	Makes MARVEL's responses more explanatory or terse.
chaining_type	Change behavior of load (admin)
display_links	Hide or Show links on screen
print_links	When printing an object, print those other object that it links to
print_back_links	When printing an object, print those other objects that link to it
prompt	Change the current Prompt (Command Line)
rule_mode	*** Unused ***
small_font	Change the small font type
normal_font	Change the normal font type
bold_font	Change the bold font type

3.1.15 unlink

The UNLINK command allows the user to remove a link between two objects in the objectbase.

A *link* is a semantic relation between two objects, as defined in the *data model*, that does not conform to the strict hierarchical composition of the objectbase. The link is rooted at one object (the source) and links to another object (the destination). To remove a link, the client must specify the source object, the link attribute, and the destination object.

1. Click on UNLINK
2. Pick the source object or DONE
3. (If not DONE) Pick the source link attribute (if there is only one link attribute, it is chosen automatically) from the provided menu
4. Pick the destination object

Errors:

If the source object selected by the client has no link attributes, MARVEL responds with "link aborted: selected object has no link attributes."

If the link attribute has no current links, MARVEL responds with "unlink: attribute <attribute> has no links."

If the destination object is not being linked to through this link attribute, MARVEL responds "<source object> is not linked to <destination object> by attribute <attibute>."

Concurrency Errors:

If another client has removed the link from <source object> to <destination object>, MARVEL responds with either "<source object> is not linked to <destination> by attribute <attribute>", or "unlink: attribute <attribute> has no links", and the screen will refresh to remove the link.

If another client has deleted either the <source object> or the <destination object>, MARVEL responds "Object Not Found. Refreshing your screen" and redisplays the current objectbase.

If another client is currently accessing the chosen object in write (exclusive) mode, then the UNLINK command will be aborted, and the following message will be printed:

> Concurrent access conflict on object <object_name>
> Your command requested accessing <object_name> in write (exclusive) mode
> But user <other client> is running the <rule> rule, which is accessing <object_name> in write (exclusive) mode
> Aborting the transaction. unlink: tx_WRITE on Current Object failed

3.1.16 usage

The USAGE command allows the user to receive usage information for the Built-In commands. When the user selects USAGE, the Usage Menu replaces the Command Menu. At this point, the user can choose to print information about each command (?), or can choose to view the information for an individual command by selecting COMMAND, and then when MARVEL prompts, the user clicks on the desired command from the Command Menu.

1. Click on USAGE
2. Click on ? to print out all the usage commands, or
   Click on COMMAND
3. Click on the command for which you wish to print out usage information.

3.2 Command Line Interface

In section 3.1, the user was expected to be using the graphical interface. MARVEL also supports a command line interface. This is useful for users who don't have access to an X-Based machine, and for creating batch sets of commands. This section will briefly provide the command line syntax required by each command:

3.2.1 add

`add -ho <name>` adds an object named <name> horizontally to the Current Object.

`add -hi <attribute> <name> [<optional-subclass name>]` adds an object named <name>hierarchically to the <attribute> set attribute of the Current Object. If the optional sub-class name is provided, the object is classified as an object of that class, only if it is a legal sub-class.

3.2.2 change

`change <name>` finds an object whose name is <name> and which has the same class as the Current Object; If found, this object is made the Current Object.

`change -c <class> <name> [ <attribute> <name> ] *` changes the Current Object to the object as directed by the user. If only <class> and <name> are entered, then this changes the Current Object to point to that object. If additional <attribute> <name> pairs are provided, then MARVEL searches vertically, through each successive object, until all pairs are exhausted.

`change -a <attribute> <name>` changes vertically to the object within the given attribute <attribute> whose name is <name>.

`change -up` changes the Current Object to be the parent of the Current Object.

3.2.3 copy

`copy <to-class> <to-name> <to-attribute>` copies the Current Object to be a child of the object of class <to-class>, and name <to-name>. The object is copied to the <to-attribute> attribute.

`copy <from-class> <from-name> <to-class> <to-name> <to-attribute>` copies the object of class <from-class>, with name <from-name>, to be a child of the object of class <to-class>, and name <to-name>. The object is copied to the <to-attribute> attribute.

3.2.4 delete

`delete <class> <name>` deletes the object from class <class> with name <name>.

3.2.5 execute

`execute <marvel-script file>` executes the specified MARVEL script file.

3.2.6 help

`help { <command> | <subject> | <rules> | ? }` gets help on the specified topic.

3.2.7 link

link <src-class> <src-name> <src-att> <dest-class> <dest-name> creates a semantic link from the object of class <src-class>, and name <src-name> to the object of class <dest-class> and name <dest-name>. The link is created in the <src-att> link attribute.

3.2.8 move

move <to-class> <to-name> <to-attribute> moves the Current Object to be a child of the object of class <to-class> and name <to-name>. The object is moved to the <to-attribute> attribute.

move <from-class> <from-name> <to-class> <to-name> <to-attribute> moves the object of class <from-class>, with name <from-name>, to be a child of the object of class <to-class> and name <to-name>. The object is moved to the <to-attribute> attribute.

3.2.9 print

print prints out the entire objectbase

print -c <class> prints out the definition of the specified class. If <class> is omitted, then all classes are printed.

print -r <name> prints out all the rules with the given name <rule>. If this parameter is omitted, then a listing of the headers of all the rules are output (not the full rule definitions).

print -C prints out information about the Current Object.

print -x <control_rule> prints out all the control rules with the given name <control_rule>. If this parameter is omitted, then a listing of the headers of all the control rules are output (not the full control rule definitions).

3.2.10 rename

rename <to-name> renames the current object to be the given name <to-name>.

rename <from-class> <from-name> <to-name> renames the object with class <from-class> and name <from-name> to the given name <to-name>.

3.2.11 set

`set [ <variable> | <variable> <value> ]*` allows the user to set certain environment variables. If the variable is a boolean flag, then no <value> required; this toggles its value. If a value is required, then the variable is set to <value>.

3.2.12 unlink

`unlink <src-class> <src-name> <src-att> <dest-class> <dest-name>` removes a link from the object of class <src-class>, and name <src-name>, to the object of class <dest-class> and name <dest-name>. The link is removed from the <src-att> link attribute.

3.3 Tips & Troubleshooting

3.3.1 Object is ambiguous in current context

When the client is attempting to fire an environment-specific rule, MARVEL sometimes responds that a given object is ambiguous. For example, suppose that the client is attempting to invoke the EDIT rule on an object named "main.c". MARVEL might respond that:

```
main.c is ambiguous in the current context:
class: CFILE, location: Top/projects/p1/src/src/modules/n/cfiles/main.c
class: CFILE, location: Top/projects/p1/src/local/modules/m/cfiles/main.c
```

When the server receives the command to EDIT "main.c", it searches from the Current Object for the closest object with this name. In this case, it found two objects named "main.c", located at different locations in the objectbase. It is the job of the user to disambiguate, so MARVEL can continue. The client would CHANGE the Current Object to the desired "main.c" (see CHANGE command), and then repeat the EDIT request.

3.3.2 Browsing during commands and rules

There are times when the objectbase becomes so dense that it is impossible to draw the names of the various objects; how is the client to know which objects are which? For example, if the client is attempting to copy an object, and the name is not displayed, how can the client determine the correct object on which to click? The answer lies in the MIDDLE button. Whenever the client clicks the MIDDLE button on an object, that object becomes the root of the sub-tree that is displayed (see BROWSE command). Figure 15 shows a simple example. Here, the client wishes to

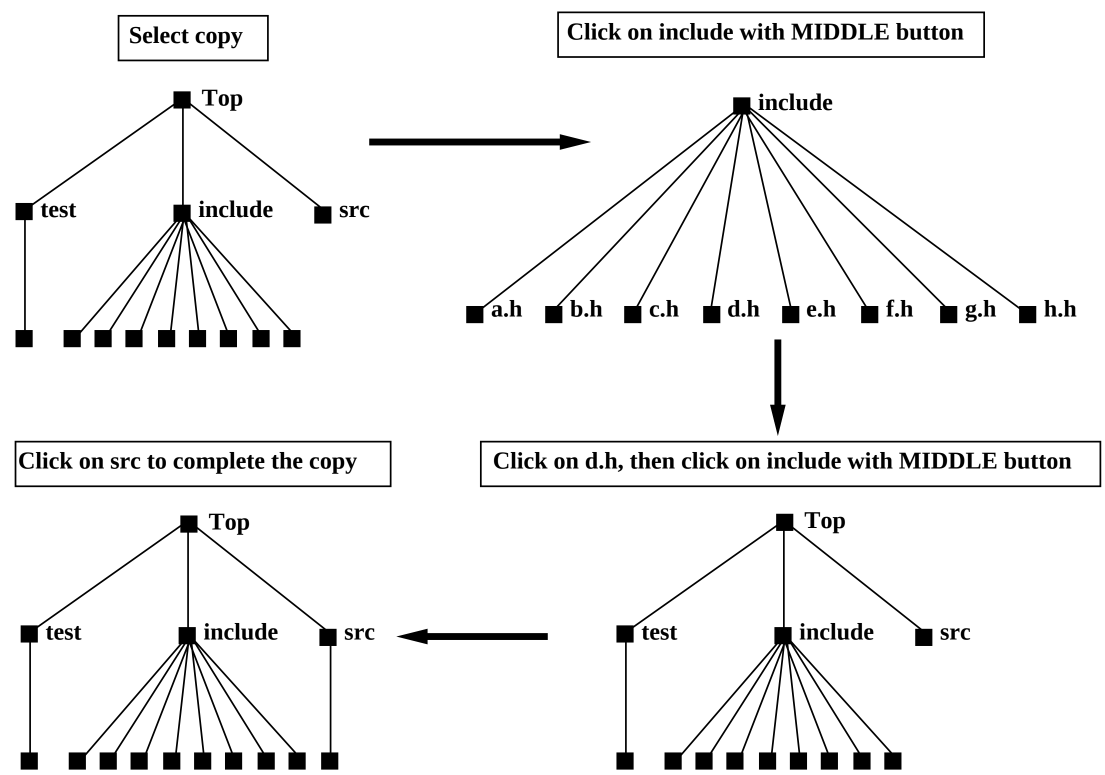

Figure 15: Browsing during commands and rules

copy the object "d.h" to be a child of "src". The client knows that the object is a child of "include", but the name is not currently visible. The client selects the COPY command and when MARVEL prompts for the source object, the client clicks on "include" with the MIDDLE button; now that "d.h" is visible, the client selects it by clicking on it with the LEFT button. MARVEL now prompts for the Target Parent Object. The client wants it to be "src", but that object is not visible, so the client clicks on "include" with the MIDDLE button, thus performing a *zoomout* (see BROWSE), and clicks on "src" with the LEFT button when it becomes visible; "d.h" is copied, and "src" now has a child "d.h". Note that the user would have to choose the set attribute to copy the object to, but that step was omitted for clarity.

3.3.3 Canceling a built-in command or rule

Most of the built-in commands have facilities for canceling once the client has selected the command. ADD, BROWSE, DELETE, LINK, PRINT, QUIT, RENAME, UNLINK all allow the client to select DONE, or CANCEL to stop the rule.

> The ADD rule only allows this when the Add Menu is displayed. Once the user has selected HIER, or HORIZ, the only way to cancel the command is to press return. MARVEL will then output the usage information for ADD.

> If the RENAME command has been invoked, and an object has already been selected by the client, the command can still be can still be canceled by pressing return when MARVEL prompts for the new name. MARVEL will reply "You must specify a name" and wait for the client to click on another object. At that point, the client can click on DONE to cancel.
>
> If the LINK or UNLINK command has been invoked, the client can cancel the command by selecting DONE. If the source object has more than one link attribute, then a Menu is displayed with these items, and the client is forced to choose one of them; thus, at that point, the command can no longer be canceled. The best that the client can do is remove (when LINK was invoked) or add (UNLINK, resp.) the link to undo the procedure.

The other commands provide no clean way for the client to cancel. And in some cases, there is no way to cancel. The CHANGE, COPY, and MOVE commands can all be canceled by making an invalid pick on the objectbase. This is done by clicking with the LEFT button in the Display Window below the last row of objects. MARVEL responds with "invalid pick", and the specific command is canceled.

The EXECUTE command prompts the client for a file name, and if the client presses return, the command is canceled.

The last three commands, HELP, SET, USAGE, have no method for canceling – the user must complete the command; for example, the user could randomly choose to click on ? if the HELP command was mistakenly selected.

An environment-specific rule can be canceled by pressing return when MARVEL prompts for the rule arguments.

3.3.4 Text Window

If the Text Window is not fully scrolled to the end when the user invokes a built-in command or rule, then the Text Window becomes garbled. Fortunately, the information is correctly printed, but the client will have to wait until the command or rule has finished before viewing what was printed. While a command or rule is executing, the Text Window cannot be scrolled. The Text Window only maintains a history of 512 lines.

While the client is executing an activity, neither the Text Window, nor the Display Window is refreshed. If either is occluded by another window, the Text Window is refreshed after every rule, and the Display Window is refreshed at the completion of all chaining.

3.3.5 Object Names

Currently, there is no restriction (other than no spaces) on the names of the objects which the client can enter. The client should take care to enter in valid names. For example, having an object named "don't" is not valid since the ' character causes problems. Other obvious names like "." and ".." should not be used.

If for some reason, an object is created with a name that causes problems, the client can still delete the object by deleting the parent object. If this object is a top level object, then the client must delete the entire objectbase manually and start again. If, however, the client has renamed an object to have a faulty name, then the client can immediately rename the object back to the original name, and even though an error message is printed, everything is back to normal.

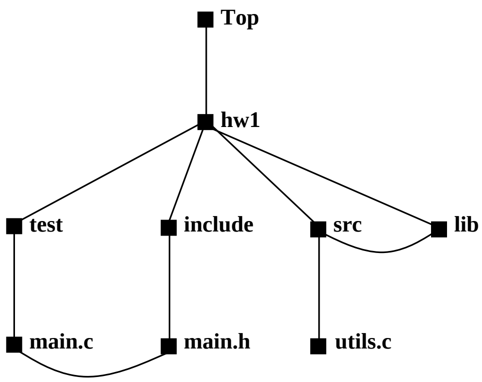

Figure 16: The Current Objectbase

4 Multi User Tutorial

The Multi-User tutorial assumes that the user has already completed the single-user tutorial in section 2. We will be using the same objectbase that the single-user tutorial creates (figure 16). We assume that the user has the server executing, and one client is running; this client will be referred to as **Client**.

4.0.6 Concurrency Conflict: Automation

We now describe a simple scenario for the Multi-User Tutorial. Assume that we want to extend the example from the single-user tutorial; specifically, we want to add several more functions and change the existing `sub1`. **Client** will change `sub1`, while **Client-2** adds some more utility functions. This tutorial *can* be simulated by one person by creating two windows, one for each client. Note: This tutorial assumes that **Client** and **Client-2** have different user id's, but are within the same UNIX group.

1. Open a window and start another **Client-2** in <**marvel_dir**>; this client decides to add another file to be a sibling of "utils.c". So **Client-2** clicks on "src" with the RIGHT button and adds hierarchically an object "new.c" to the *cfiles* attribute by typing "cfiles new.c". Note that the objectbase for **Client** has not changed to display this object. To update the objectbase at any time, a client may select the REFRESH command, and if **Client** does this then the most recent objectbase will be displayed. **Client-2** should go ahead and edit "new.c" and add the following routine. As before, MARVEL will execute backward chaining to RESERVE the file. **Client-2** should not exit `emacs`.

```
#include <stdio.h>

void sub2()
{
```

```
  printf("sub2: Beginning\n");
  printf("Now calling sub1\n");

  sub1();
  printf("sub2: End\n");
}
```

2. While **Client-2** is working, **Client** should click on "new.c" to see how MARVEL displays concurrent information. The last line of this output states "(locked by user <user> in mode X)". This tells us that <user> is accessing this object in Exclusive Mode. Click on "src" with the LEFT button and observe the lock description for this object: "(locked by user <user> in mode IX)". This tells us that <user> is accessing a descendant of this object (new.c) in Exclusive Mode.

3. **Client** should edit "utils.c" and modify the `sub1` routine to be the following:

```
#include <stdio.h>

void sub1()
{
  printf("sub1: Beginning\n");

  printf("sub1 not implemented.\n");

  printf("sub1: End\n");
}
```

4. **Client** should now save these changes and exit `emacs`. Figure 17 contains the result of this command. There were two concurrency conflicts reported. The first occured when MARVEL attempted to fire the DIRTY rule on the parent MODULE of "utils.c". You should print out the DIRTY rule, and look for DIRTY[MODULE]. Note that the CONDITION checks through all the children CFILE objects of the given MODULE for an object that satisfies the two given predicates. Since **Client-2** is currently accessing "new.c" in Exclusive Mode, this condition is unable to be satisified, so the DIRTY rule is not fired. Similar reasoning explains the second concurrency conflict when MARVEL forward chains to ARCH "utils.c".

5. Have **Client-2** exit `emacs` now, and execute the BUILD rule on "test" to build the test program.

4.0.7 Concurrency Conflict: Consistency

There is a second type of conflict that arises which causes a transaction to be rolled back. For further information on transactions see the Administrator's manual. Essentially, when a forward automation chain fails, the transaction is aborted, and

Client Window

```
Now reserving .../Top/projects/hw1/src/src/cfiles/utils.c/utils.c
Now creating RCS directory...
.../Top/projects/hw1/src/src/cfiles/utils.c/RCS/utils.c,v <--
    .../Top/projects/hw1/src/src/cfiles/utils.c/utils./c
initial revision: 1.1
enter description, terminated with single '.' or end of file:
NOTE:  This is NOT the log message!

contains useful functions
.

>> >> done
.../Top/projects/hw1/src/src/cfiles/utils.c/utils.c reserved

Editing utils.c
Changes Made and saved.
analyzing utils.c on Mon Sep 16 18:35:51 EDT 1991

lint utils.c
analysis successful, results available with viewErr
compiling utils.c on Mon Sep 16 18:35:56 EDT 1991

compile successful, results available with viewErr
```

Marvel Text Window

```
FIRING RULE : edit (utils.c)
  PRECONDITIONS for edit( utils.c)
  Can backward chain to ...
   reserve( utils.c )

  EXECUTING: reserve( utils.c )

  EXECUTING: reserve( utils.c )
  EXECUTION RESULT : reserve( utils.c ) has satisfied precondition of edit (utils.c )
  RE-EVALUATING: edit(utils.c)

  EXECUTING : edit( utils.c )

  EXECUTING : edit( utils.c )
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        edit(utils.c)
The following locking conflict could not be resolved:

Concurrent access conflict on new.c
    Your command fired the dirty rule, which requested accessing new.c in
    read (shared) mode. But user <user> is running the edit rule, which is
    accessing new.c in write (exclusive) mode.
Aborting the transaction.
Can forward chain to ...
    analyze(utils.c)

  EXECUTING analyze( utils.c )
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        analyze(utils.c )
  Can forward chain to ...
    compile(utils.c )

  EXECUTING compile( utils.c )
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                        compile(utils.c )
The following locking conflict could not be resolved:

Concurrent access conflict on new.c
   Your command fired the dirty rule, which requested accessing new.c in
   read (shared) mode.  But user <user> is running the edit rule, which is
   accessing new.c in write (exclusive) mode.
Aborting the transaction.
Execution cycle completed
```

Figure 17: Multi User Concurrency Conflict

MARVEL continues the forward chaining cycle. For example, when the DIRTY rule was aborted, MARVEL was still able to ANALYZE and COMPILE in figure 17. However, if a consistency forward chain fails, then MARVEL must "rollback" changes to restore the objectbase to a consistent state.

1. Have **Client-2** execute the EXEC_PROG rule on "test". When **Client-2** presses <return> in the Client Window (in response to "Enter in the parameters for .../testexec"), the program outputs the following:

   ```
   Version: 1.0
   This is the main file.
   sub1: Beginning
   sub1 not implemented.
   sub1: End
   ```

2. Now, have **Client-2** execute the EXEC_PROG rule on "test" again, but don't press <return> in the Client Window.

3. **Client** should click on "test" with the LEFT button to view the status for that object: "(locked by user <user> in mode X)".

4. Have **Client** edit "utils.c" and change the definition of `sub1` to be:

```
#include <stdio.h>

void sub1()
{
  printf("sub1: Beginning\n");

  printf("sub1 has a syntax error\n")

  printf("sub1: End\n");
}
```

Please note the missing semicolon after the second `printf` statement. **Client** should save these changes and exit `emacs`. Figure 18 contains the output of the following forward chaining cycle. MARVEL now initiates the forward chaining cycle and fires the ANALYZE rule, which succeeds, and the COMPILE rule, which fails. As a result of this failure, MARVEL chains to the DIRTY rule on "src" and then fires the DIRTY rule on "lib". No problems have occured so far. However, after MARVEL executes the DIRTY rule on "lib", it forward chains, through a consistency chain, to the DIRTY rule on PROJECT "hw1". This rule can't be fired, since **Client-2** is currently accessing "test" in exlusive mode. So MARVEL must rollback these consistency chains which, in this case, are the the two DIRTY rules.

5. **Client** can verify that these DIRTY rules were indeed rolled back by printing out the information for "src" and "lib". In both cases, the archive_status has been restored to "Archived."

Marvel Text Window

```
FIRING RULE : edit (utils.c)
  EXECUTING : edit (utils.c)

  EXECUTING : edit (utils.c)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                          edit(utils.c)
Can forward chain to ...
    analyze(utils.c)

  EXECUTING analyze(utils.c)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                       analyze(utils.c)
  Can forward chain to ...
    compile(utils.c)

  EXECUTING compile(utils.c)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                       compile(utils.c)

  EXECUTING: dirty(src)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                             dirty(src)

  EXECUTING: dirty(lib)
  EVALUATING FORWARD CHAINING POSSIBILITIES AFTER EXECUTING:
                                                              dirty(lib)
The following locking conflict could not be resolved

Concurrent access conflict on object test
    Your command fired the dirty rule, which requested accessing test in
    read (shared) mode.  But user <user> is running the exec_prog rule,
    which is accessing test in write (exclusive) mode.
Aborting the transaction.
Rolling Back dirty
Rolling Back dirty
Consistency chain is rolled back.

Execution cycle completed
```

Figure 18: Multi User Concurrency Conflict With Rollback

5 C/Marvel Sources

This section contains all the strategy files that make up the Tutorial as presented in this manual. It also includes all the SEL shell scripts that enact each of the tools. This is a valuable resource for learning how rules work together to form chains, and how SEL shell scripts perform their accomplished tasks

5.1 Strategy Files

5.1.1 cmarvel.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      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.

strategy cmarvel

# Import all the addition data and process models needed to build up the
# environment.

imports data_model, dirty, archive, build, compile, execute, doc, edit,
rcs, print;

exports all;
```

5.1.2 data_model.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      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.  A Marvel objectbase can contain several group objects.

GROUP :: superclass ENTITY;
    build_status : (Built, NotBuilt, Initialized) = Initialized;
    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;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    build_status : (Built, NotBuilt, Initialized) = Initialized;
    libraries : set_of LIB;
    programs : set_of PROGRAM;
    doc : set_of DOC;
    incs : set_of INC;
    src  : set_of MODULE;
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;
    build_status : (Built, NotBuilt, Initialized) = Initialized;
    build_log : text;

    docs : set_of DOC;
    cfiles : set_of CFILE;
    incs : set_of INC;
    exec : binary;
end


# LIB is a shared archive type library. The ultimate representation of a
# library is a .a file, that is, an archive format file.

LIB :: superclass ENTITY;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    afile : binary = ".a";
```

```
end


# MODULE organizes CFILES based upon some higher order.  Each module knows
# which library (possibly more than one) it will be archived to.  MODULES
# can recursively contain other MODULES, and sets of CFILES.

MODULE :: superclass ENTITY;
    library : set_of link LIB;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    modules : set_of MODULE;
    cfiles : set_of CFILE;
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;
    owner : user;
    timestamp : time;
    reservation_status : (CheckedOut, Available, Initialized) = Initialized;
    contents : text;
end

# Extra information is needed to record the state of compilation and
# analysis (lint, in our case) for CFILEs.  A CFILE contains links to
# various HFILEs that it #includes.

CFILE :: superclass FILE;
    compile_status : (Compiled, NotCompiled, Initialized) = Initialized;
    compile_log    : text;
    analyze_status : (Analyzed, NotAnalyzed, Initialized) = Initialized;
    analyze_log    : text;

    contents : text = ".c";
    object_code : binary = ".o";
    ref : set_of link HFILE;
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 : boolean = false;
    contents : text = ".h";
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 : boolean = false;
    formatted_file : binary;
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;
    documents : set_of DOCUMENT;
    files : set_of DOCFILE;
end


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

DOCUMENT :: superclass ENTITY;
    docfiles : set_of DOCFILE;
end


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

INC :: superclass ENTITY;
    archive_status : (Archived, NotArchived, Initialized) = Initialized;
    hfiles : set_of HFILE;
end


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

BIN :: superclass ENTITY;
```

```
    executable : binary;
end

end_objectbase
```

5.1.3 dirty.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#

# This strategy contains various "dirty" procedures to make sure that
# if a certain file is modified, then certain propogations take place.
# These rules are hidden from the user since the user should not be
# allowed to use these rules.

strategy dirty

imports data_model;
exports all;

rules

hide dirty[?l:LIB]:
   (exists MODULE ?m suchthat (linkto [?m.library ?l]))
   :
   [?m.archive_status = NotArchived]
   {}
   [?l.archive_status = NotArchived];


hide dirty[?m:MODULE]:
   (exists CFILE ?c suchthat (member [?m.cfiles ?c]))
   :
   (or [?c.analyze_status = NotAnalyzed]
       [?c.compile_status = NotCompiled])
   {}
   [?m.archive_status = NotArchived];
```

```
hide dirty[?Proj:PROJECT]:
   (and (forall PROGRAM ?P suchthat (member [?Proj.programs ?P]))
(forall LIB ?L suchthat (member [?Proj.libraries ?L])))
   :
   (or [?L.archive_status = NotArchived]
       [?P.build_status = NotBuilt])
   {}
   (?Proj.build_status = NotBuilt);


hide clean[?proj:PROJECT]:
   (forall LIB ?l suchthat (member [?proj.libraries ?l])):

   [?l.archive_status = Archived]
   {}
   [?proj.archive_status = Archived];


hide dirty[?g:GROUP]:
   (exists PROJECT ?proj suchthat (member [?g.projects ?proj]))
   :
   [?proj.archive_status = NotArchived]
   {}
   no_forward (?g.build_status = NotBuilt );


hide clean[?g:GROUP]:
   (forall PROJECT ?proj suchthat (member [?g.projects ?proj]))
   :
   [?proj.build_status = Built]
   {}
   no_forward (?g.build_status = Built);

hide clean[?h:HFILE]:
   (and (exists INC ?i suchthat (member [?i.hfiles ?h]))
        (exists PROJECT ?proj suchthat (member [?proj.incs ?i])))
   :
   (and ( ?proj.build_status = NotBuilt )   # otherwise cleans right after
        ( ?h.recompile_mod = true ))     # dirtying the files.

   {}
   [ ?h.recompile_mod = false ];
```

5.1.4 archive.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      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

imports data_model;
exports all;

objectbase

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

end_objectbase

rules

# This rule archives a module if all its CFILES have been compiled.
#
arch[?m:MODULE]:

    (and (forall CFILE ?c suchthat (member [?m.cfiles ?c]))
 (exists LIB   ?l suchthat (linkto [?m.library ?l])))
    :
    (?c.compile_status = Compiled)

    { ARCHIVER archive ?c.object_code ?l.afile }

    [?m.archive_status = Archived];
    [?m.archive_status = NotArchived];
```

```
## 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[?l:LIB]:
    (forall MODULE ?m suchthat (linkto [?m.library ?l]))
    :
    [?m.archive_status = Archived]
    { }
    [?l.archive_status = Archived];


# list_arch: this rule just lists the contents of an archive.
#
list_arch[?l:LIB]:
   :
   { ARCHIVER list_archive ?l.afile }
   ;
```

5.1.5 build.load

```
#
#                   Marvel Software Development Environment
#
#                                Copyright 1991
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#

strategy build

# This strategy provides a rule to build a PROGRAM.  It also provides two
# inferences rules that will build an entire PROJECT or GROUP.

imports data_model;
exports all;

objectbase

BUILD :: superclass TOOL;
    build_program : string = build;
end
```

```
end_objectbase

rules


# Build the group only if every project has been built
#
build [?group:GROUP]:
    (forall PROJECT ?proj suchthat (member [?group.projects ?proj]))
    :
    [?proj.build_status = Built]
    { }
    no_forward (?group.build_status = Built);


# Build the project only if all the programs are built and all the
# libraries and include files have been archived.
#
build [?proj:PROJECT]:
    (forall PROGRAM ?p suchthat (member [?proj.programs ?p])):
    [?p.build_status = Built]
    { }
    no_forward ( ?proj.build_status = Built);


# Build a program if all the C files of the program have been successfully
# analyzed and compiled.

build[?p:PROGRAM]:
    (and (forall PROJECT ?P suchthat (member [?P.programs ?p]))
         (forall MODULE  ?m suchthat (member [?P.src ?m]))
 (forall LIB     ?l suchthat (linkto [?m.library ?l]))
         (forall CFILE   ?c suchthat (member [?p.cfiles ?c])))
    :
    (and (?c.compile_status = Compiled)
         (?l.archive_status = Archived))

    { BUILD build_program ?c.object_code ?p.exec ?p.build_log ?l.afile }

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

5.1.6 compile.load

```
#
#                   Marvel Software Development Environment
#
#                                Copyright 1991
#                     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;
    analyze : string = analyze;
end

end_objectbase

rules

compile [?c:CFILE]:

    (forall HFILE ?h suchthat (linkto [?c.ref ?h])):

    # 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 ( ?c.analyze_status = Analyzed )
         no_backward ( ?c.compile_status = NotCompiled))


    { COMPILER compile ?c.contents ?c.compile_log ?c.object_code ?h.contents }
```

```
    ( ?c.compile_status = Compiled );
    [ ?c.compile_status = NotCompiled ];

analyze[?c:CFILE]:

    (forall HFILE ?h suchthat (linkto [?c.ref ?h])):

    no_backward (?c.analyze_status = NotAnalyzed)

    { COMPILER analyze ?c.contents ?c.analyze_log ?h.contents }

    ( ?c.analyze_status = Analyzed);
    [ ?c.analyze_status = NotAnalyzed ];
```

5.1.7 execute.load

```
#
#                   Marvel Software Development Environment
#
#                                Copyright 1991
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#

strategy execute

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

imports data_model;
exports all;

objectbase

EXECUTE :: superclass TOOL;
    exec : string = execute;
end

end_objectbase

rules

exec_prog[?p:PROGRAM]:
```

```
    :
    no_forward (?p.build_status = Built)

    { EXECUTE exec ?p.exec "RUN" }
    ;

debug[?p:PROGRAM]:
    :
    no_forward (?p.build_status = Built)

    { EXECUTE exec ?p.exec "DEBUG" }
    ;
```

5.1.8 doc.load

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# doc envelope:  Contains rules for formatting the given document, and for
# displaying the document in xdvi.
#

strategy doc

imports data_model;
exports all;

objectbase

     FORMAT :: superclass TOOL;
         format_latex : string = format_latex;
         display_dvi  : string = display;
     end

end_objectbase

rules

display[?doc:DOCFILE]:
```

```
   :
   (?doc.reformat_doc = false)

   { FORMAT display_dvi ?doc.formatted_file }
   ;

format[?doc:DOCFILE]:
   :
   (?doc.reformat_doc = true)

   { FORMAT format_latex ?doc.contents ?doc.formatted_file }

   (?doc.reformat_doc = false);
   (?doc.reformat_doc = true);
```

5.1.9 edit.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      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 which displays
# 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;
exports all;

objectbase

EDITOR :: superclass TOOL;
    edit : string = editor;
    edit_h : string = editor_h;
end

VIEWER :: superclass TOOL;
```

```
    viewErr : string = viewErr;
    viewBuildErr : string = viewBuildErr;
    view    : string = view;
end

end_objectbase

rules

# 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[?c:CFILE]:

    # if the file has been reserved, you can go ahead and edit it
    :
    (and ( ?c.owner = CurrentUser )
 ( ?c.reservation_status = CheckedOut))


    { EDITOR edit ?c.contents ?c.analyze_status ?c.analyze_log
  ?c.compile_status ?c.compile_log }


    (and (?c.analyze_status = NotAnalyzed)
 (?c.compile_status = NotCompiled)
         (?c.timestamp = CurrentTime));
    no_chain ( ?c.reservation_status = CheckedOut );


# this edit rule is for editing document files.

edit[?f:DOCFILE]:

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

    { EDITOR edit ?f.contents }

    (and (?f.reformat_doc = true)
 (?f.timestamp = CurrentTime));
```

```
    no_chain ( ?f.reservation_status = CheckedOut );


# this edit rule is for editing include files.

edit[?h:HFILE]:

    # if the file has been reserved, you can go ahead and edit it
    :
    (and ( ?h.owner = CurrentUser )
 ( ?h.reservation_status = CheckedOut))

    { EDITOR edit_h ?h.contents }

    (and (?h.recompile_mod = true)
 (?h.timestamp = CurrentTime));
    no_chain ( ?h.reservation_status = CheckedOut );


# The following rule views output from the compiler and analyzer for a
# particular file.

viewErr[?f:CFILE]:
    :
    { VIEWER viewErr ?f.analyze_log ?f.compile_log }
    ;

viewErr[?p:PROGRAM]:
    :
    { VIEWER viewBuildErr ?p.build_log }
    ;

view[?f:FILE]:
    :
    { VIEWER view ?f.contents}
    ;
```

5.1.10 rcs.load

```
#
#                    Marvel Software Development Environment
#
```

```
# This strategy contains rules for doing revision control on FILE type objects.

strategy rcs

imports data_model;
exports all;

objectbase

RCS  :: superclass TOOL;
    reserve : string = check_out;
    create  : string = create_rcs;
    deposit : string = check_in;
end

end_objectbase

rules

# reserve:  reserve a file type object.  In the C/Marvel example, you can
#           use this rule on FILE, CFILE, HFILE and DOCFILE, because of
#           the inheritance mechanism.

reserve[?f:FILE]:
    :
    (or [ ?f.reservation_status = Available ]
        [ ?f.reservation_status = Initialized ])

    { RCS reserve ?f.contents }

    (and no_forward ( ?f.reservation_status = CheckedOut )
 no_forward ( ?f.owner = CurrentUser ));
    no_chain ( ?f.reservation_status = Available );

# 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 = CheckedOut ])

    { RCS deposit ?f.contents }

    no_forward ( ?f.reservation_status = Available );
    no_chain ( ?f.reservation_status = CheckedOut );
```

5.1.11 print.load

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#

strategy print

# This strategy contains rules to print out various files.
#

imports data_model;
exports all;

objectbase

PRINT :: superclass TOOL;
    print_hp : string = print_hp;
    print_dvi : string = print_dvi;
    choose_printer : string = print_choose;
end

end_objectbase

rules

Print [?f:DOCFILE]:
  :
  (?f.reformat_doc = false)
```

```
  { PRINT print_dvi ?f.formatted_file}
  ;

Print [?c:CFILE]:
  :
  { PRINT print_hp ?c.contents }
  ;

Print [?h:CFILE]:
  :
  { PRINT print_hp ?h.contents }
  ;
```

5.2 SEL Shell Scripts

5.2.1 analyze

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# analyze envelope
#
# usage: analyze [CFILE.contents] [CFILE.analyze_log] [HFILE...]
#
ENVELOPE

SHELL ksh;

INPUT
  text         : thefile;
text      : log_file;
        set_of HFILE : hfiles;
OUTPUT
none ;

BEGIN

echo "analyzing `basename $thefile` on `date`"
echo
```

```
echo "analyzing `basename $thefile` on `date`" > $log_file
echo >> $log_file
echo >> $log_file

# we need to make the -I list
#----------------------------
tmp_dir=/tmp/analyze$$
mkdir $tmp_dir

# Now the header files
#---------------------
for i in $hfiles
do
  ln -s $i $tmp_dir
done

echo "lint `basename $thefile`"
lint $thefile -I$tmp_dir >> $log_file 2>&1
LINTSTATUS=$?

# Remove the Temporary Directory
# ------------------------------
if [ "x$tmp_dir" != "x" ]
then
   rm -r $tmp_dir
fi

if [ $LINTSTATUS -eq 0 ]
then
    echo analysis successful, results available with viewErr
    echo analysis successful >> $log_file
    RET_CODE=0
else
    echo analysis failed, results available with viewErr
    echo analysis failed >> $log_file
    RET_CODE=1
fi

RETURN "$RET_CODE" ;
END
```

5.2.2 archive

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# archive envelope
#
# This archives the given object code file(s) into the specified
# library(ies).  Apparently, Under the "ibmrt" platform, ranlib has
# to be run every time an object code is archived.  Is this the case?
#
# usage: archive [set_of FILE.object_code] [set_of LIB.afile]
#

ENVELOPE

SHELL ksh;

INPUT
set_of cfiles : obj_code;
set_of binary : library;
OUTPUT
none ;
BEGIN

FINAL_RET_CODE=0
for lib in $library
do
  echo running ar rv on `basename $lib.`
  echo
  ar rv $lib $obj_code
  ARCHIVE_STATUS=$?

  RET_CODE=1

  if [ $ARCHIVE_STATUS -eq 0 ]
  then
    if [ $MT != "ibmrt" ]
    then
        echo running ranlib on $arch
```

```
        ranlib $lib
        RANLIB_STATUS=$?
    fi

    if [ $RANLIB_STATUS -eq 0 ]
    then
        echo archive now available in `basename $lib`
RET_CODE=0
    fi
  fi

  if [ $RET_CODE -eq 1 ]
  then
     FINAL_RET_CODE=1
  fi
done

RETURN "$FINAL_RET_CODE" ;
END
```

5.2.3 build

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# build envelope
#
# usage: build [CFILE.object_code] [PROGRAM.exec] [LIB.afile]
#

ENVELOPE

SHELL sh;

INPUT
set_of binary : cfiles;
binary : executable;
text : build_log;
set_of binary : afiles;
```

```
OUTPUT
none ;

BEGIN
   echo "build $executable on `date`"
   echo "build $executable on `date`" > $build_log

   # we need to make the -I list
   # ---------------------------
   tmp_dir=/tmp/analyze$$
   mkdir $tmp_dir

   echo "cc $cfiles $afiles -ll -lc -lm -o $executable"
   echo "cc $cfiles $afiles -ll -lc -lm -o $executable" >> $build_log
   cc $cfiles $afiles -ll -lc -lm -o $executable >> $build_log 2>&1

   # this checks for existence, and to be sure
   # it is the proper kind of executable.
   # ------------------------------------
   MT=`arch`

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

   if [ $ans -eq 0 ]
   then
       if ( test -x $executable )  # if the file is executable, then the
```

```
       then      # build was successful.
  echo build successful
  RET_CODE=0
       else
  echo build failed      # Otherwise, notify the user that
  RET_CODE=1      # since the executable bit was not set
       fi
   else
       echo build failed
       RET_CODE=1
   fi

RETURN "$RET_CODE" ;
END
```

5.2.4 check_in

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# check_in envelope
#
# usage: check_in [FILE.contents]
#

ENVELOPE check_in;

SHELL sh;

INPUT
  text         : contents;
OUTPUT
none ;

BEGIN

BASENAME=`basename $contents`
DIR=`dirname $contents`
rcsdirectory=$DIR/RCS
```

```
echo "Check in $contents [y/n]?"
read REPLY

if [ "x$REPLY" = "xy" ]
then

   # Get Message from the User
   # -------------------------
   echo "Please Enter a Message:"
   read REPLY

   # the -f option forces a deposit even if not different from
   # previous version.  This is done to avoid an annoying INPUT/OUTPUT
   # problem between ci and the shell.  If you want to see it, try it
   # without the -f option.
   if ( test -n "$REPLY" )
   then
      ci -f -m"$REPLY" -u $rcsdirectory/${BASENAME},v $contents
   else
      ci -f -u $rcsdirectory/${BASENAME},v $contents
   fi

   # Always a tricky subject: Remove the file,
   # and check it back out.
   # -----------------------------------------
   rm -f $1/$object
   co $rcsdirectory/${BASENAME},v $contents

   echo $contents deposited
   RET_CODE=0
else
   echo $contents NOT deposited
   RET_CODE=1
fi

RETURN "$RET_CODE" ;
END
```

5.2.5 check_out

```
#
#                    Marvel Software Development Environment
#
```

```
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# check_out envelope
#
# usage: check_out [CFILE.contents]
#

ENVELOPE

SHELL sh;

INPUT
  text         : contents;
OUTPUT
none ;

BEGIN

BASENAME=`basename $contents`
DIRNAME=`dirname $contents`
rcsdirectory=$DIRNAME/RCS

echo Now reserving $contents

# create the RCS directory if it doesn't already exist.
# ---------------------------------------------------
if [ ! -d $rcsdirectory ]
then
   mkdir $rcsdirectory
   echo Now creating RCS directory...
fi

if [ ! -f $rcsdirectory/$BASENAME,v ]
then
   # If this file hasn't been deposited yet, we must do it the first
   # time.  Note, that we retain the lock through the -l option
   # ----------------------------------------------------------------
   if [ ! -f $contents ]
   then
      touch $contents
   fi
```

```
   ci -l $rcsdirectory/${BASENAME},v $contents
   echo $contents reserved
   RET_CODE=0
else
   echo "check out $contents [y/n]? "
   read REPLY

   if [ "x$REPLY" = "xy" ]
   then
      co -l $rcsdirectory/${BASENAME},v $contents
      echo $contents reserved
      RET_CODE=0
   else
      echo "Reservation cancelled at your request."
      echo $contents not reserved
      RET_CODE=1
   fi
fi

RETURN "$RET_CODE" ;
END
```

5.2.6 compile

```
#
#                    Marvel Software Development Environment
#
#                                  Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# compile envelope
#
# usage: compile [CFILE.contents]
#  [CFILE.compile_error]
#  [CFILE.object_code]
#  [HFILE...]
#

ENVELOPE

SHELL ksh;
```

```
INPUT
  text         : thefile;
text      : error_file;
        binary       : thebinary;
        set_of HFILE : hfiles;
OUTPUT
none ;

BEGIN

echo "compiling `basename $thefile` on `date`"
echo

echo "compiling `basename $thefile` on `date`" > $error_file
echo >> $error_file
echo >> $error_file

# we need to make the -I list
#--------------
tmp_dir=/tmp/compile$$
mkdir $tmp_dir

# Now the header files
#--------------
for i in $hfiles
do
  ln -s $i $tmp_dir
done

echo "cc -g -c -ll -lc -lm -lX11 -I$tmp_dir -in: $thefile -out:$thebinary" >> $error_file
cc -g -c -I$tmp_dir $thefile -ll -lc -lm -lX11 -o $thebinary >> $error_file 2>&1
CCSTATUS=$?

if [ "x$tmp_dir" != "x" ]
then
   rm -r $tmp_dir
fi

if [ $CCSTATUS -eq 0 ]
then
    echo compile successful, results available with viewErr
    echo compile successful >> $error_file
    RET_CODE=0
else
    echo compile failed, results available with viewErr
```

```
    echo compile failed >> $error_file
    RET_CODE=1
fi

RETURN "$RET_CODE" ;
END
```

5.2.7 display

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# Display Envelope: This copies the dvi file to ~/Marvel/cmarvel/pix.dvi
# where a previous process is currently reading it.
#
# This assumes the following shell variable is defined.
# MARVEL_PIC=$MARVEL_ROOT/pix.dvi

# display envelope
#
# usage: display [DOCFILE.formatted_file]
#

ENVELOPE

SHELL ksh;

INPUT
        binary       : thedvi;
OUTPUT
none ;
BEGIN

echo "Now copying file $thedvi to $MARVEL_PIC"
cp $thedvi $MARVEL_PIC

xdvi $thedvi &

RETURN "0" ;
```

```
END
```

5.2.8 editor

```
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# editor envelope
#
# usage: edit [CFILE.contents] [CFILE.analyze_status] [CFILE.analyze_log]
#             [CFILE.compile_status] [CFILE.compile_log]
#
# This edits the chosen file, and incorporates a simple locking mechanism
# by making the file writable when it edits it, and removing this capability
# when it leaves.  Note: If there are analyzed or compilation errors, then
# this script will add those to the emacs buffer.  (I don't use vi --
# someone else can find out how to edit multiple buffers in vi.)

ENVELOPE editor;

SHELL sh;

INPUT
text : thefile;
enum : analyze_status;
text : analyze_log;
enum : compile_status;
text : compile_log;

OUTPUT
none ;

BEGIN

BASENAME=`basename $thefile`
echo
echo Editing $BASENAME.

# Determine if the file is there already
# ----------------------------------------
```

```
Created="YES"
SaveReport=`ls -l $thefile`
if [ $? -eq 0 ]
then
   Created="NO"
fi


# Edit the file.  Check to make sure on an X Terminal.
#
T=`echo $EDITOR | cut -d' ' -f1`
if [ "x$T" = "xemacs" ]
then
     # Create an emacs .el load file
     # ----------------------------
     EL_FILE=/tmp/editor$$.el
     touch $EL_FILE

     # Check to see if we need to add analyze_log/compile_log
     # buffers to the editing process.
     # ------------------------------
     if [ $compile_status = "NotCompiled" ]
     then
        echo \(find-file-other-window \"$compile_log\"\) >> $EL_FILE
     fi

     if [ $analyze_status = "NotAnalyzed" ]
     then
echo \(split-window\) >> $EL_FILE
        echo \(find-file \"$analyze_log\"\) >> $EL_FILE
     fi

     emacs -fn 9x15 -geometry 80x55 $thefile -l $EL_FILE

     # Remove the Editor specific file
     # ------------------------------
     rm $EL_FILE
else
     vi $thefile
fi

# Check to make sure that the file really existed.
#
if [ $Created = "YES" ]
then
```

```
   echo "File `basename $thefile` Created."
   RET_CODE=0
else
   NewReport=`ls -l $thefile`

   if [ "$SaveReport" = "$NewReport" ]
   then
      echo "No Changes Made"
      RET_CODE=1
   else
      echo "Changes Made and saved."
      RET_CODE=0
   fi
fi

RETURN "$RET_CODE";
END
```

5.2.9 editor_h

```
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# editor_h envelope
#
# usage: edit_h [HFILE.contents]
#
# This edits the chosen file, and incorporates a simple locking mechanism
# by making the file writable when it edits it, and removing this capability
# when it leaves.

ENVELOPE editor;

SHELL sh;

INPUT
text : thefile;

OUTPUT
none ;
```

```
BEGIN

BASENAME=`basename $thefile`
echo
echo Editing $BASENAME.

# Determine if the file is there already
# --------------------------------------
Created="YES"
SaveReport=`ls -l $thefile`
if [ $? -eq 0 ]
then
   Created="NO"
fi


# Edit the file.  Check to make sure on an X Terminal.
#
T=`echo $EDITOR | cut -d' ' -f1`
if [ "x$T" = "xemacs" ]
then
     emacs -fn 9x15 -geometry 80x55 $thefile
else
     vi $thefile
fi



# Check to make sure that the file really existed.
#
if [ $Created = "YES" ]
then
   echo "File `basename $thefile` Created."
   RET_CODE=0
else
   NewReport=`ls -l $thefile`

   if [ "$SaveReport" = "$NewReport" ]
   then
      echo "No Changes Made"
      RET_CODE=1
   else
      echo "Changes Made and saved."
      RET_CODE=0
```

```
   fi
fi

RETURN "$RET_CODE";
END
```

5.2.10 execute

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# execute envelope
#
# usage: execute [PROGRAM.exec] { debug DEBUG/RUN }
#
ENVELOPE

SHELL ksh;

INPUT
        binary : thebinary;
literal : debug;

OUTPUT
 none ;

BEGIN

echo "executing $thebinary on `date`"
echo

#
# Get Parameters for execution
#

echo "Enter in the parameters for $thebinary: "
read -r
if ( test $REPLY )
```

```
then
  ARGUMENTS=""
else
  ARGUMENTS=$REPLY
fi

#
# Execute or Debug on this executable with above parameters
#

if [ $debug = "DEBUG" ]
then
  dbx -s "dbx.alias" -r $thebinary $ARGUMENTS
else
  $thebinary $ARGUMENTS
fi

RETURN "0";
END
```

5.2.11 format_latex

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# format_latex envelope
#
# usage: format_latex [DOCFILE.contents] [DOCFILE.formatted_file]
#
ENVELOPE

SHELL ksh;

INPUT
  text         : contents;
        binary       : thebinary;
OUTPUT
none ;
```

```
BEGIN

echo "Now processing $contents"

BASENAME=`getname $contents`

latex $contents
if [ $! -eq 0 ]
then
echo "Cleaning up temp files"
rm $BASENAME.aux
rm $BASENAME.log
mv $BASENAME.dvi $thebinary
RET_CODE=0
else
echo "Errors in latex"

# Remove binary, since it is now out of date.
#
if [ test -f $thebinary ]
then
  rm $thebinary
fi
RET_CODE=1
fi

RETURN "$RET_CODE" ;
END
```

5.2.12 list_archive

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# list_archive envelope
#
# usage: list_archive [LIB.afile]

ENVELOPE list_archive;
```

```
SHELL ksh;

INPUT

binary : library;

OUTPUT

none ;

BEGIN

echo "`basename $library` contains the following modules:"
ar t $library | sort

RETURN "0" ;
END
```

5.2.13 print_dvi

```
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# Envelope: print_dvi
#
# Usage   : print_dvi [DOCFILE.formatted_file]
#

ENVELOPE print_dvi;

SHELL ksh;

INPUT
text : formatted_file;
OUTPUT
none ;
```

```
BEGIN

  echo "Now Printing $formatted_file to $PRINTER"
  dvips $formatted_file

RETURN "0";
END
```

5.2.14 print_hp

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# Envelope: print_hp
#
# Usage   : print_hp [FILE]

ENVELOPE print_hp;

SHELL ksh;

INPUT
text : file;
OUTPUT
none ;
BEGIN

  echo "Now Printing $file to $PRINTER"
  lpr $file

RETURN "0";
END
```

5.2.15 view

```
#                     Marvel Software Development Environment
#
```

```
#                                Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# view envelope
#
# usage: view [FILE.contents]


ENVELOPE

INPUT
text : thefile;

OUTPUT
none ;

BEGIN

echo
echo Viewing $thefile ...

if [ "x$EDITOR" = "x" ]
then
     less $thefile
else
     xterm -e less $thefile &
fi

RETURN "0" ;
END
```

5.2.16 viewBuildErr

```
#
#                    Marvel Software Development Environment
#
#                                Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# viewBuildErr envelope
```

```
#
# usage: viewBuildErr [PROGRAM.build_log]
#

ENVELOPE

SHELL ksh;

INPUT
text : build_log;

OUTPUT
none ;

BEGIN

echo "======================== build errors ========================"
$PAGER $build_log

RETURN "0" ;
END

\End{verbatim}

\subsubsection{viewErr}
\begin{verbatim}
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# viewErr envelope
#
# usage: viewErr [CFILE.analyze_log] [CFILE.compile_log]
#

ENVELOPE

SHELL ksh;

INPUT
text : analyze_log;
```

```
text : compile_log;

OUTPUT
none ;

BEGIN

echo "======================= analyze errors ========================"
cat $analyze_log

echo "!===================== compile errors ======================="
cat $compile_log

RETURN "0" ;
END
```

6 Marvel References

The Marvel project began in June 1986 jointly between Prof. Gail Kaiser of Columbia University and Dr. Peter Feiler of the Software Engineering Institute, when Prof. Kaiser was a Visiting Computer Scientist at the SEI. The first implementation was done during the fall of 1986 by Steve Popovich, then a staff member at the SEI, by modifying the SMILE environment developed at Carnegie Mellon University as part of Dr. Nico Habermann's Gandalf project during the early 1980's. This was the version 0 implementation of Marvel. Kaiser, Feiler and Popovich had all been members of the Gandalf project. The research effort was inspired by concepts introduced in SMILE and in the CommonLisp Framework being developed by Dr. Bob Balzer's group at the Information Sciences Institute. Marvel was named after Professor Marvel, the name of the Wizard of Oz character in his Kansas incarnation.

The Marvel project moved to Columbia University, and the first serious implementation work began in January 1987. PhD student Naser Barghouti headed the effort involving numerous project students: Russel Goldberg, Joe Milligan, Michael Sacks, Tam Tran, Wendy Dilliard, Christine Hong, Wai Keung Hui, and Alexander Mogieleff. During the summer of 1987, Barghouti and Kaiser conducted frequent discussions with Dr. Bob Schwanke of Siemens Corporate Research, leading to refinements of the initial Marvel concepts. This work resulted in Marvel version 1.0, which was also implemented on top of SMILE.

The implementation of the more robust Marvel 2.x, independent of SMILE, began in September 1987 with project students Qifan Ju, Christine Lombardi and Mike Sokolsky. Later, the system was almost entirely rewritten through the joint efforts of Naser Barghouti and Mike Sokolsky, initially as an MS thesis student and starting in September 1988 as a research staff associate. Marvel was by now strongly influenced by the "process programming" concept promoted by Prof. Lee Osterweil of the University of California at Irvine and other members of the Arcadia consortium, as well as by ongoing discussions at the annual International Software Process Workshop series. Marvel 2.01 was demonstrated at the 3rd ACM Practical Software Engineering Environments conference in November 1988.

Marvel 2.10 was used in several class projects for the E6123 Programming Environments and Software Tools course, involving Laura Johnson, Victor Kan, Kok-Yung Tan and Michael Tannenblatt. There were further contributions by project students Neil Arora, Issy Ben-Shaul, Laura Johnson, David Robinowitz, Miriam Sporn, Kok-Yong Tan, and Michael Tannenblatt. Ari Shamash worked on the Marvel user interface as a part-time employee during Summer 1989. Project student Mara Cohen collaborated with Barghouti and Sokolsky on the Marvel user manual. The Marvelizer tool was added to make it possible to immigrate existing software into a Marvel environment.

The first Marvel release was version 2.5, in spring 1989. Marvel 2.6 soon followed, in June 1989. Marvel 2.5 or 2.6 was licensed to 15 sites: IBM, Siemens

Corporate Research, University of Arizona, University of Maryland, Imperial College (United Kingdom), University of Pisa (Italy), Software Design & Analysis, Software Research Associates (Japan), Instep, University of Nancy (France), University of Victoria (Canada), Purdue University, Kestrel Institute, Bell-Northern Research (Canada), and Digital Equipment Corporation. MARVEL 2.6 was demonstrated at the ACM International Conference on Management of Data in May 1990. A special version, MARVEL 2.65, was developed at the instigation of Software Design & Analysis, which has been using MARVEL as a platform for investigating the implementation of their activity structures process modeling formalism.

Work on the first multiple-user MARVEL, version 3.0, began in summer 1990. This effort was led by Issy Ben-Shaul, first as an MS thesis student and beginning in September 1990 as a research staff associate, together with PhD students Naser Barghouti, George Heineman and Mark Gisi, and part-time employee Tim Jones. A preliminary version was used in E6123 in spring 1991 for class projects by all students: Tim Jones, Kui Mok, Tushar Patel, Ari Shamash, Chikuei James Show, and Bruce Zenel. During summer 1991, Will Marrero contributed as a part-time employee, and PhD student Steve Popovich joined the project. Preliminary versions were demonstrated at the 13th International Conference on Software Engineering in May 1991, and at Software Design & Analysis and the 6th Knowledge-Based Software Engineering conference in September 1991. MARVEL 3.0 is scheduled for release in October 1991.

Work has already begun on MARVEL 3.1, to incorporate Barghouti's PhD thesis results and other improvements. Kevin Lam participated in summer 1991 as a part-time employee, and there are currently several visitors and project students expected to contribute during fall 1991. Release is expected in spring 1992.

At one time or another during the MARVEL project, funding was provided by NSF Presidential Young Investigator Award CCR-8858029, NSF Research Instrumentation grant CDA-8920080, and NSF grant CCR-9106368; by AT&T Foundation Special Purpose Grants, a DEC Incentives for Excellence award, a General Electric fellowship, an IBM Research Initiation Grant, and an IBM Fellowship; grants from Digital Equipment Corporation, Bell-Northern Research, Siemens Corporate Research, SRA America, Sun Microsystems, and Xerox Foundation; the NSF Engineering Research Center for Telecommunications and a focal project of the New York State Center for Advanced Technology - Computer & Information Systems. In addition to the above, the Programming Systems Laboratory at Columbia University was funded by NSF grants CCR-8802741 and CCR-900930; grants from Citibank Financial Markets Group, two IBM Fellowships, and IBM contracts and joint studies; a NYS CAT seed project, NASA training grant NGT 50583, and an American Association of University Women dissertation fellowship.

The following publications and dissertations have directly resulted from the MARVEL project:

- Gail E. Kaiser and Peter H. Feiler. An Architecture for Intelligent Assistance in Software Development. *Ninth International Conference on Software Engineer-*

ing, March 1987, pp. 180-188.

- Peter H. Feiler and Gail E. Kaiser. Granularity issues in a knowledge-based programming environment. *Information and Software Technology*, Butterworth Scientific, 29(10):531-539, December 1987.

- Naser S. Barghouti and Gail E. Kaiser. Implementation of a Knowledge-Based Programming Environment. *Twenty-first Hawaii International Conference on System Sciences*, January 1988, volume II, pp. 54-63.

- Gail E. Kaiser, Peter H. Feiler and Steven S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 5(3):40-49, May 1988.

- Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler and Robert W. Schwanke. Database Support for Knowledge-Based Engineering Environments. *IEEE Expert*, 3(2):18-32, Summer 1988.

- Gail E. Kaiser and Naser S. Barghouti. An Expert System for Software Design and Development. Invited paper in *Joint Statistical Meetings*, August 1988, pp. 10-19.

- Michael H. Sokolsky, *Data Migration in an Object-Oriented Software Development Environment*, MS thesis, Columbia University, CUCS-424-89, April 1989.

- Gail E. Kaiser, Naser S. Barghouti and Michael H. Sokolsky. Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel. *Twenty-third Hawaii International Conference on System Sciences*, January 1990, vol. II, pp. 131-140.

- Naser S. Barghouti and Gail E. Kaiser. Modeling Concurrency in Rule-Based Development Environments. *International Working Conference on Cooperating Knowledge Based Systems*, Springer-Verlag, October 1990, pp. 223-239.

- Naser S. Barghouti and Gail E. Kaiser. Multi-Agent Rule-Based Software Development Environments. *Fifth Knowledge-Based Software Assistant Conference*, September 1990, pp. 375-387.

- Naser S. Barghouti and Gail E. Kaiser. Modeling Concurrency in Rule-Based Development Environments. *IEEE Expert*, 5(6):15-27, December 1990.

- Israel Z. Ben-Shaul. *An Object Management System for Multi-User Programming Environments*. MS thesis, Columbia University, CUCS-010-91, April 1991.

- George T. Heineman, Gail E. Kaiser, Naser S. Barghouti and Israel Z. Ben-Shaul. Rule Chaining in Marvel : Dynamic Binding of Parameters. *Sixth Annual Knowledge-Based Software Engineering Conference*, September 1991, pp. 276-287.

- Naser S. Barghouti and Gail E. Kaiser. Scaling Up Rule-Based Development Environments. To appear in *Third European Software Engineering Conference*, October 1991.

- Mark A. Gisi and Gail E. Kaiser. Extending A Tool Integration Language. To appear in *First International Conference on the Software Process*, October 1991.

- Michael H. Sokolsky and Gail E. Kaiser. A Framework for Immigrating Existing Software into New Software Development Environments. To appear in *Software Engineering Journal*, Michael Farraday House, November 1991.

- Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD Thesis, Columbia University, expected November 1991.

The following publications and dissertations are tangentially related to the MARVEL project:

- Gail E. Kaiser and Peter H. Feiler. Intelligent Assistance without Artificial Intelligence. *Thirty-Second IEEE Computer Society International Conference*, February 1987, pp. 236-241.

- Dewayne E. Perry and Gail E. Kaiser. Models of Software Development Environments. *Tenth International Conference on Software Engineering*, April 1988, pp. 60-68.

- Calton Pu, Gail E. Kaiser and Norman Hutchinson. Split-Transactions for Open-Ended Activities. *Fourteenth International Conference on Very Large Data Bases*, August 1988, pp. 26-37.

- Shyhtsun F. Wu. *(Towards a Framework For Comparing Object-Oriented Systems*. MS thesis, Columbia University, CUCS-438-89, July 1989.

- Gail E. Kaiser. A Marvelous Extended Transaction Processing Model. *Eleventh World Computer Conference IFIP '89*, Elsevier Science Publishers B.V., August 1989, pp. 707-712.

- Gail E. Kaiser. AI Techniques in Software Engineering. In Hojjat Adeli, ed., *Knowledge Engineering, Vol. II, Applications*, McGraw-Hill, 1990, ch. 7, pp. 213-244.

- Gail E. Kaiser. A Flexible Transaction Model for Software Engineering. *Sixth International Conference on Data Engineering*, February 1990, pp. 560-567.

- Gail E. Kaiser. Interfacing Cooperative Transactions to Software Development Environments. *Office Knowledge Engineering*, IEEE Computer Society Technical Committee on Office Automation, 4(1):56-78, February 1991.

- Gail E. Kaiser and Dewayne E. Perry. Making Progress in Cooperative Transaction Models. *Data Engineering*, 14(1):19-23, March 1991.
- Dewayne E. Perry and Gail E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering*, 17(3):283-295, March 1991.
- Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. To appear in *ACM Computing Surveys*, September 1991.
- Gail E. Kaiser and Calton Pu. Dynamic Restructuring of Transactions. To appear in Ahmed K. Elmagarmid, ed., *Database Transaction Models for Advanced Applications*, chapter 8, Morgan Kaufmann, 1991.

Gail E. Kaiser
Associate Professor of Computer Science
Columbia University in the City of New York
4 October 1991