Marvel 3.0
Administrator's Manual

Programming Systems Laboratory
Columbia University
450 Computer Science Building
New York, NY 10027
(212) - 854-2736
fax: (212) - 666-0140

TR CUCS–032-91

October 9, 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

Preface

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

1 Introduction

This manual is neither a user's manual for Marvel, nor an implementor's manual; it has no information on how to use Marvel or how to modify the source code. We assume the reader has a working knowledge of Marvel and software development environments in general. If this is the first time you use Marvel, it is strongly recommended to follow the user's guide tutorial once the basic installation steps were made (see section 2), rather then launching into writing an environment without prior experience with using Marvel .

1.1 Who Would Use This Manual

There are basically three groups of people who interact with Marvel. The first are end users who use the system for development purposes. They should refer to the user's manual.
The second group of people are implementors who want to modify the kernel code. They should refer to the implementor's manual.
The third group of people are the system/project administrators, for whom this manual is directed. A project administrator writes specifications of the project's process, data, and consistency models, which are loaded into Marvel and tailor its behavior accordingly.

1.2 Conceptual Overview

1.2.1 Research Goals

The long-term goal of the Marvel project is to develop a kernel for multi-user software development environments that allows teams of programmers to cooperate on developing a large-scale software project. The kernel provides concurrency control and object management primitives that 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.

1.2.2 Concepts

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, process model and consistency model. All specifications are written in the Marvel Strategy Language (MSL). The administrator loads these specifications into the kernel, creating a Marvel environment that supports the data management and consistency, and the process enaction 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. Attribute types include primitive types, files, single and set of contained objects, and single and set of directed links. Set attributes contain instances of other classes as their values, thus implementing *composite objects*, and giving the Marvel object management system (OMS) a hierarchical traversal capability. Links are typed and point to any instances in the objectbase thus giving the Marvel OMS arbitrary graph traversal capability. 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 and envelopes. Rules specify the behavior of the tailored Marvel environment in terms of what commands are available and what kind of assistance is provided. Envelopes interfaces between the tools/activities and Marvel rules and objects. 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.

The consistency is defined by the administrator in terms of consistency predicates in the rules and in terms of default values as defined in the class definitions.

1.2.3 Selected Publications

There are numerous publications about Marvel for reference. A selected list follows: (the full list is presented in the preface).

The concept of Marvel and the implementation details of the single-user version are covered in [7, 12, 2, 8, 13].
Our experience using single-user Marvel is documented in [6].
The concept of the multi-user version of Marvel and the problems encountered when scaling up the system are described in [3, 4, 5].
Our experience integrating external tools through SEL, our shell envelope language, are described in [10].

Further work on the rule processor, in particular the dynamic binding of parameters during chaining, is described in [9].

1.3 System Overview

The Marvel system is based on a client/server architecture[5], where the clients communicate with the server via tcp/ip sockets. A Marvel server can support zero or more clients sharing access to the same objectbase. When some user starts a client, a special daemon (marveld) installed in the operating system (/etc/inetd.conf on SunOS and Ultrix) checks whether or not there is already a server running for that objectbase. If so, it connects them up and if not it brings up a server on the same machine as the client. When all its clients have quit, the daemon shuts the server down. The daemon triggers the installed version of the Marvel server. A server can also be started manually (e.g., to test new versions of the server or if the daemon is not installed in your system). Only one server can execute for a given objectbase at a time, and a special file (.server_port) in the objectbase indicates whether or not there is a server currently running and provides information that clients use internally to connect to this server.

A Marvel objectbase is stored persistently in a particular file system directory, known as a "Marvel environment". A Marvel environment is a directory that contains a set of Marvel Strategy Language (MSL) files, a set of envelopes, a binary objectbase, an internal representation of the contents of the MSL files, a "hidden" file system for binary and text file attributes, and some additional subdirectories and files for things like failure recovery logs and maintaining a persistent counter for generating clientIDs. The binary representation of the in-memory objectbase is stored in a gdbm[1] file, while file attributes (text and binary) are stored in a "hidden" file system. This file system includes directories representing objects containing set attributes (see section 3.6 for more details).

All Marvel environments are entirely independent of each other, and there is no identifier resolution of any sort across environments. Marvel environments are set up by "administrators", charged with defining appropriate data organization and behavior for a (class of) software project, and the typical end-user need know nothing about their contents. The administrator usually loads a set of MSL files into a Marvel environment, and thereafter end-users simply use that environment without further recourse to the **load** command.

A Marvel client corresponds one-to-one with an operating system process. A client is a very important concept with respect to Marvel 's implementation. Every client executes as a "session" running from its invocation to its exit, where the session provides various client-specific information such as the controlling user's Unix environment variables. A client, i.e., a session, is in the middle of zero or one rule chain at a time. A Marvel server keeps a "context" for each session, and performs rule

[1]gdbm is a package for efficient storage/retrieval of byte streams on disk

chaining with respect to a session.

When a client is not in the midst of chaining, this does not mean it is inactive; a client may also execute *built-in* commands. This includes **load** (the data, process, and consistency models), **quit** (exit the client), **help** (obvious), a large number of commands for browsing the objectbase (including commands related to various display options), and another large number of commands for directly modifying the objectbase (**add**, **delete**, **move**, **copy**, etc.).

There is currently no ability for a client to run anything in the background except through envelopes (using the normal facilities of the operating system to fork new processes). It is possible, however, for an envelope to invoke a new Marvel client in batch mode, feeding it a script to execute, and terminate. It is not possible to create a client in this manner and hand it off to a human controller. Thus batch clients cannot become interactive.

A Marvel end-user may have multiple clients for the same server running under his userid, either on the same or different machines. If the graphical front end is used, then multiple clients for the same server belonging to the same user would correspond to multiple windows. The Marvel graphical user interface for a single client appears to have several subwindows, but from the X11 windows point of view, there is exactly one window. The internal windows are manipulated by Marvel itself and cannot be manipulated using conventional X11 facilities. Even the top-level window does not respond normally to X11 user commands provided by the user's choice of window manager (because the Marvel graphical user interface is implemented directly in Xlib). An end-user can also interact with multiple clients through the command line interface, by using operating system commands to move them between the background and foreground, or separate user jobs controlled by multiple terminals (or X windows xterms).

A Marvel user corresponds one-to-one with an operating system userid. If multiple humans are logged in under the same userid, then they appear to be the same user. The notion of a user is not an important concept from Marvel 's viewpoint. Its only distinction is as a built-in type (and the corresponding CurrentUser variable) for use by environment administrators in writing classes and rules. Marvel itself does not distinguish between clients controlled by the same versus different users.

The Marvel server currently supports a fairly conventional scheduling mechanism among the clients. The scheduler does not round robin time slices among the clients, but instead feeds off a queue of messages from clients. The messages might indicate either a new command (either built-in or a rule) or the completion of an activity. The server takes a message from the queue. If it is a built-in, it executes the command atomically – even if it is an extremely long duration command, notably Marvelizer[13].

Marvel 3.0's concurrency control mechanism supports relatively conventional serializability among multiple users. Rules are subtransactions and rule chains are nested transactions. Two-phase locking is used to enforce serializability. The main difference from standard mechanisms is that a rule or rule chain never blocks waiting for a lock, but instead is either terminated or aborted if it cannot acquire the necessary locks. Thus if two activities attempt to acquire conflicting locks, one will be aborted or terminated, depending on which reflect consistency and/or automation. (The distinction between these will be explained in section 3.4).

MARVEL currently runs on Sun 3s (running SunOS 4.0.3), Sun 4s (SunOS 4.1.1) and DecStation 3100s (Ultrix 3.1; an old version, note the most recent version distributed by Digital is something like 4.2). There are no restrictions on mixing and matching clients, i.e., clients and server can run on same or different machine and or same or different architecture. However, an objectbase created by a server executing on a particular architecture can only be accessed later by servers executing on the same architecture because of binary incompatibility problems. However, a binary objectbase can be manually converted from one architecture to another using special utility programs called bin2ascii and ascii2bin utilities (see appendix C).

MARVEL 3.1 (expected to be released around January 1992) will support relaxation of serializability as defined by the administrator in the *coordination model*, specified in terms of control rules, which describe how to handle conflicts among particular kinds of rules. Refer to [1] for an extensive discussion of these issues.

1.3.1 System Components

Figure 1 illustrates the internal structure of MARVEL. For more information refer to the implementor's manual and to [5].

The server is composed of the following modules:

- *Storage Manager (SM)* - This bottom layer provides persistent storage and controls the flow of data from main to secondary storage. This layer has no knowledge of the data model and therefore can be easily replaced. SM provides services for, and communicates with, the OM. It is built on top of the gdbm package.

- *File Manager (FM)* - This module manages the interface between the OM and the "hidden" file system, by providing system calls to access files on the file system.

- *Object Manager (OM)* - This module implements the object-oriented data model. The upper layers deal only with the object abstraction. OM communicates with SM to transform the raw data into objects, and provides services to LM. SM, FM, and OM together comprise the Data Management part of MARVEL (DM).

- *Lock Manager (LM)* - This layer, usually considered as part of the transaction manager, is treated as a separate layer, to support the complete separation between transactions and object management. LM is the *conflict-detection* layer, and it serves as a mediator between TM and OM and provides the handles

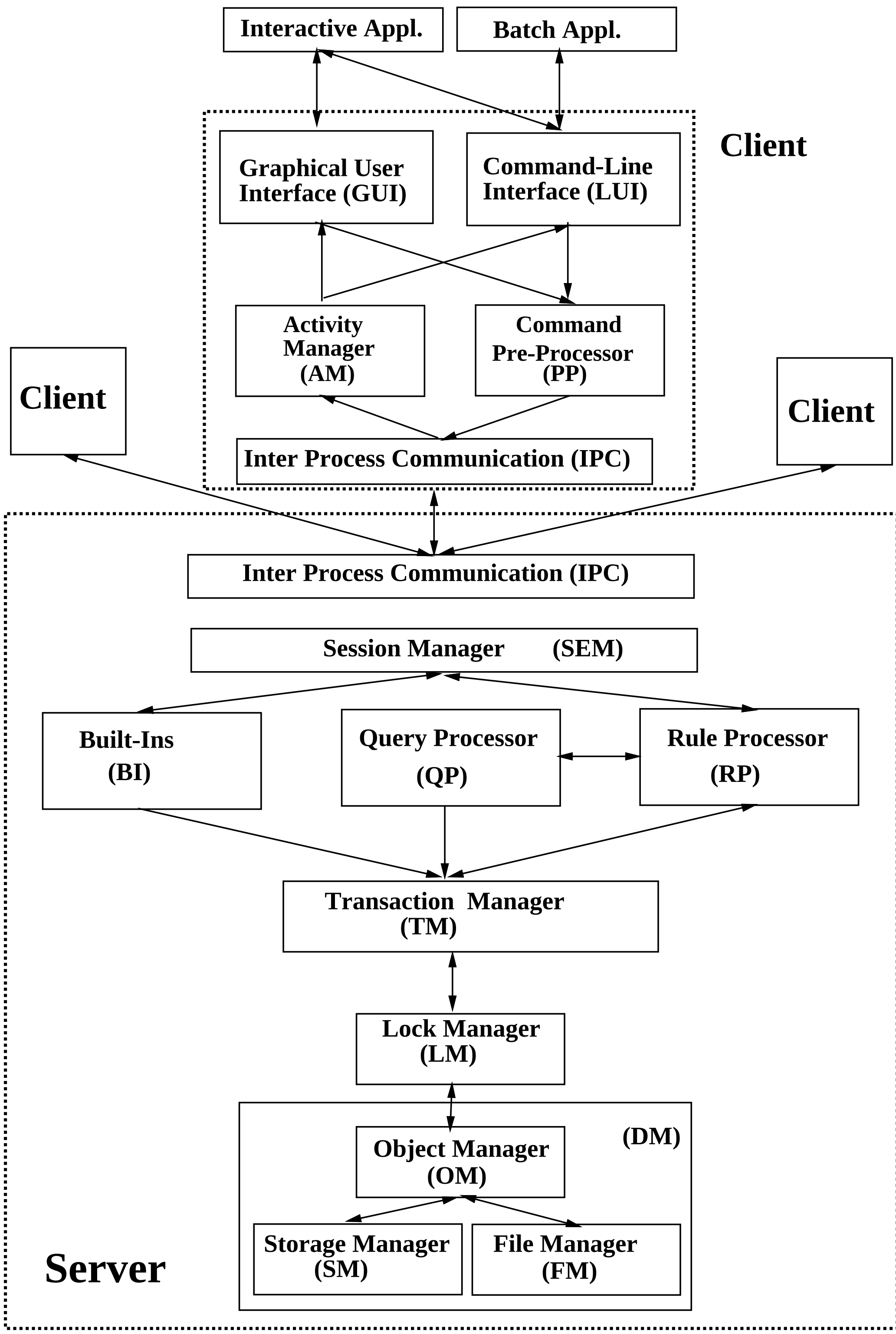


Figure 1: Marvel 3.0 Architecture

for the transaction manager to enforce its policy. LM reads in a file (compatibility.matrix) that specifies the lock modes and their compatibility. Thus, modifying the contents of the matrix can be done without recompilation of the code. Modifying the dimensions of the matrix (by adding lock modes) can also be done but then TM has to be augmented with support for the new locks.

- *Transaction Manager (TM)* - This layer is responsible for controlling concurrent access to the data and maintain its consistency. It does so by enforcing concurrency control, transaction recovery, and crash recovery policies. TM communicates with LM when it needs to access the objectbase, and is called from RP and QP.

- *Query Processor (QP)* - This layer translates the queries expressed in a query language into calls to the TM. The QP is accessible only through RP at the moment.

- *Rule Processor (RP)* - A central part of the system, it processes requests for rule firing and implements the chaining engine to enact a specific process. It assumes the set of rules that were loaded into MARVEL using the load command (see section 5).

- *Built-Ins (BI)* - A set of kernel commands available from any environment.

- *Inter Process Communication (IPC)* - This layer provides the communication between the client application and the server, and is responsible for receiving and sending messages. Like OM, IPC must preserve the object abstraction when transferring objects between the client and the server via a sequential medium.

- *Command Pre-Processor (PP)* - This layer does pre-processing of built-in commands and other requests to the user before sending them to the server.

- *Activity Manager (AM)* - This module manages the activities run in the client and interfaces them to MARVEL through envelopes.

- *User Interface (UI)* - This is the uppermost layer of MARVEL . It interfaces between the human user and the environment. It provides both graphical and command-line interface. The latter is mainly used for batch processing, using the **execute** command. An important component of the graphical UI is a display of an image of the objectbase and its structure.

2 Installation

This section outlines the installation procedure required in order to be able to run Marvel. It does not describe the procedure for creating or setting up a specific environment. This is described in detail in section 3 and in the user's guide. Basic familiarity with Unix is assumed throughout this section.

2.1 Unloading the Distribution Tape

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

1. Select a userid. If you unload the tape as root, all the files will probably be owned by a user not on your system. Otherwise, the user who unloads the tape will own all the files. The file system must have at least 75 Megabytes of free storage. (large parts of what comes in the distribution tapes does not have to reside on disk, but it would be a good idea to at least unload everything once and inspect the contents.)

2. Find an appropriate tape drive. The Sun tapes were made on a standard Sun drive on a sun4. The DEC tapes were made on a TK50

3. `cd` to a directory under which you desire to have the MARVEL release reside. There must not exist a directory or file called `marvel` there, and it must have write permissions.

4. Insert the distribution tape into the tape drive

5. Unload the tape with tar:

   ```
   tar xvf /dev/rst0 > tar.out
   ```

   When done, look at the file tar.out and verify that all files were unloaded properly and remove tar.out.

6. Remove the tape

2.2 Contents of the Distribution Tape

The directory listing in the new directory should consist of the following:

1. `README.setup` - Setup procedure to start up a MARVEL session.

2. `administrators` - This file contains userids of people who are allowed to log in to MARVEL as administrators. You should modify it to include the administrators in your group.

3. `bin` - all binaries and shell scripts that are used in order to maintain and run MARVEL live here. There are executables for all architectures but you should not be concerned with those since all programs have shell front-ends that invoke the appropriate executable.

4. `compatibility.matrix` - This file defines the default locking policy in 3.0. Do NOT modify this file unless you are fully acquainted with the concurrency control mechanism in MARVEL (refer to the implementor's manual for further information).

5. `doc` - This directory includes all the documentation on-line.

6. `environments` - This directory contains several example environments for reference. In particular, `Marvel.3.0` is an environment that contains all the sources of MARVEL 3.0.

7. `flatsrc` - All sources are arranged here in a flat directory. Useful for using `tags` to browse through the sources.

8. `help` - online help files used by MARVEL.

9. `src` - the complete source tree.

10. `lib` - libraries for compilation.

11. `buglist` - files listing all known bugs in MARVEL.

2.3 Installing the MARVEL Daemon

The MARVEL daemon (marveld) is an optional program that starts up and terminates a MARVEL server on demand. If not installed, MARVEL can still be used, but the server has to be manually started and terminated. You might want to try running MARVEL without installing the daemon and install it at a later point, once you are familiar with manual operation of the server.

The installation of the server daemon requires modifying some system configuration files and restarting a system daemon. If you're not an experienced system administrator, you're probably best off asking one at your site to help you with this.

Installing the server daemon requires the addition of a line to `/etc/group`, `/etc/inetd.conf`, `/etc/passwd`, and `/etc/services` on each machine that you'll want to run a server on. If you're using NIS (YP) with a "group", "passwd" and/or "services" map, you need to modify the appropriate map(s) on the NIS master.

1. Set MARVELHOME and PROJECT to be shell environment variables denoting the MARVEL system directory where you installed the system.

2. Become root on the system

3. Pick an unused IP port for the daemon to listen to. This should be some number between IPPORT_RESERVED and IPPORT_USERRESERVED (defined in /usr/include/netinet/in.h on most systems) that does not already appear in /etc/services. Call this number X.

4. Add the following line to /etc/services (or the "services" map on the NIS master):

   ```
   marvel X/tcp # Marvel SDE server daemon
   ```

5. Create a group named `marvel` by adding a line similar to the following to /etc/group (or the "group" map on the NIS master):

   ```
   marvel:*:x:peter,paul,mary
   ```

   (Replace the `x` with some unused gid at your site. Also, replace `peter, paul, and mary` with the userids of the people who will be using marvel at your site.)

6. Create an account named `marvel` by adding a line similar to the following to `/etc/passwd` (or the "passwd" map on the NIS master):

   ```
   marvel:*:y:x:Marvel SDE:/dev/null:/dev/null
   ```

   (Replace the `y` with some unused uid at your site. Use the same value for `x` as in step 6.)

7. Add the following line to `/etc/inetd.conf`:
   For Sun systems:

   ```
   marvel stream tcp nowait marvel $MARVELHOME/bin/marveld
   ```

   For DEC systems:

   ```
   marvel stream tcp nowait $MARVELHOME/bin/marveld marveld
   ```

   If you're installing this on another platform, you'll have to figure out how the entry should read by looking at the man page for inetd.conf. To help you out, here are some values for the most common fields:

   - `service-name = marvel`
   - `socket-type = stream`
   - `protocol = tcp`
   - `wait-status = nowait`
   - `user-id = marvel`
   - `server-program = $MARVELHOME/bin/marveld`
   - `argv[0] = marveld` (not required on some systems, such as Sun)

8. Restart inetd by sending a HUP signal to it. One way of doing this is by:
   ```
   kill -HUP `ps ax|grep 'inetd'|grep -v 'grep'|cut -d' ' -f3`
   ```

From now on, every time you run MARVEL on a specific environment, the daemon will start the server for you if there is no server running already.

3 MSL Programmer's Guide

This section outlines the MSL language in full, provides detailed examples of all the constructs with full explanations, and provides the source for the C/Marvel environment, complete with full documentation. It is intended for those who are planning to design and write environments for Marvel.

3.1 Introduction

In order to learn how to write and design environments in Marvel, one has to understand the main concepts that comprise a coherent environment and how to customize it to the project's own needs. This is a multi-step process that is best described by detailing each of the following subjects:

1. *data model*
2. *process model*
3. *assistance and consistency model*

Throughout this manual we will be using the C/Marvel environment as our primary example; Appendix B, page 64 contains the full source code for this environment. However, the methods described in this manual can be generalized to any environment.

This section ends with a description of the contents of the "environment" directory that stores all the environment-specific information.

3.2 Data Model: An Object-Oriented Approach

The first step in designing an environment is to provide an organization of its components. In Marvel, these specifications are described in terms of a set of object-oriented (OO) class definitions. The reader is expected to be familiar with this paradigm, and the following concepts are described with reference to Marvel.

3.2.1 Objects

An object is the basic component in the data model; it is a sequence of bytes in memory that has a defined *class* and a set of *attributes*. As in other object-oriented systems, every object has a name, a unique object identifier, and a state, denoted by the values of its attributes. In addition, every object is persistent. Persistence is handled by the Object Management System (OMS) component of Marvel. (More

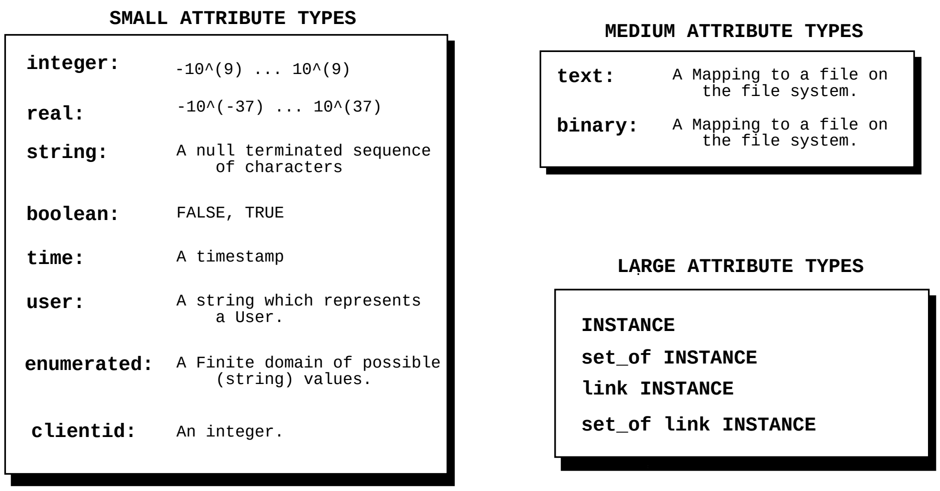


Figure 2: Attribute types in Marvel

information about the OMS can be obtained in the implementor's manual). The behavior of an object, usually expressed with associated methods, is determined by rules that manipulate it (see section 3.4). However, rules may apply to multiple objects, thus they are treated as multi-methods.

3.2.2 Classes

A class defines a set of *objects*, and specifies the *attributes* that each object has. In Marvel, an attribute is defined as one of the following four types: *small, medium, large,* and *link* attributes.

Small attributes denote the state of an object and can be formed from a set of primitive types. The current set of primitive types consists of integer, real, boolean, string, enumerated and three special attributes namely *user*, *clientid*, and *time*. The *user* type corresponds to a Unix userid. In addition to verifying that a given string maps to a real userid in the system, Marvel provides special operators for manipulating this attribute, namely CurrentUser, which returns the userid of the owner of the client process that is currently served by the Marvel server, and ResetUser, which resets the user field to NULL value. The *clientid* type corresponds to the unique client identifier given by the system at login time. It is mostly used to provide access control to objects at the client level. (Note that a user can have multiple clients running at the same time so the *user* type is not sufficient for controlling each client separately.) The *clientid* type can be accessed only by the CurrentClient and ResetClient operators, with the same semantics as the analogous *user* operators described above. *time* corresponds to an internal representation of the time. It can be manipulated only by the CurrentTime operator, which returns the system time.

Medium attributes map to files in the "hidden" file system. In order to provide "black-box" integration of tools, Marvel provides an interface from the object-based OMS

```
# 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
```

Figure 3: Class Definition in MSL

to the file-based tools. (See section 4 for explanation of the interface.) The OMS itself maps the request for files needed by the tool into the corresponding file attributes. Note that an object can map to zero, one, or more files/directories in the file system. There are two kinds of file attributes: text and binary.

The large attributes represent a containment relationship among objects, thus creating the *composite-object hierarchy*. The composition hierarchy is an important concept in MARVEL, since it allows to abstract the project's components via composition. When designing the data model for a project, the administrator has to identify carefully how to devise the project into sub-components that can be handled independently. Two types of large attributes exist: single and set_of. The former allows only one child of that type to be created, while the latter allows arbitrary number of objects to created as children.

Finally, link attributes allow the data model to maintain arbitrary semantic connections between two objects in the objectbase, outside of the composite-object hierarchy. One major difference between link and large attributes is that an object connected via large attribute to its parent is considered to be part of that object. This means that deleting an object implies deleting all its children connected via large attributes, but the linked objects are not affected. Like in large attributes, there are two types of link attributes, namely single and set. Figure 3 shows an example of a class definition in MSL .

In addition to data-object classification, MSL provides another type of class, for definition of tools that represent the external activities that MARVEL allows the rules to use. However, these classes are used merely for the definition of tools and not for instantiation. Figure 4 shows an example of a tool class definition. In here, `COMPILER` is a class of tools that are semantically similar. There are three TOOL_METHODS, *compile*, *lex_compile*, and *yacc_compile*, that operate on objects of class CFILE, LFILE, and YFILE respectively. Each tool_method is a string attribute which maps to an envelope on the file system. The tool interface is described in more detail in section 3.4.5.

```
COMPILER :: superclass TOOL;
        compile        : string = compile_c;
        lex_compile    : string = compile_lex;
        yacc_compile   : string = compile_yacc;
end
```

Figure 4: Tool Class Definition

Appendix B.1, page 64, contains a graphical depiction of the composite-object hierarchy for the C/MARVEL environment. The reader should examine this now, for some examples in this manual refer to this. Note how a given class can appear more than once in the hierarchy, at different levels.

3.2.3 Inheritance

MARVEL provides for inheritance, that is, one can define a subclass/superclass relationship among classes. A subclass usually denotes a specialization property, i.e., a subclass has additional properties besides the properties defined by its superclass. A subclass can denote specialization in either the data or the behavior or both. Behavioral aspects are dealt in section 3.4. For now it suffices to say that behavioral properties are also inherited.
For example, if class C_1 has attributes att_a and att_b, C_2 has attribute att_1, and class C_2 is defined to be a superclass of C_1, then C_1 inherits the att_1 attribute. Inheritance is a recursive process, so in this example, C_1 inherits any attributes that C_2 also inherited. MARVEL also allows for multiple-inheritance, so a class can inherit attributes from a set of classes. In case a class inherits an attribute that is defined in multiple superclasses, the first superclass (as defined in the list of superclasses) has the highest precedence. Every data class in MARVEL is defined to be a subclass of the ENTITY superclass. Tool classes are defined as subclasses of the TOOL class but there is no inheritance mechanism with respect to tool classes. Figure 5 is an example of inheritance relationship between the `CFILE` class and the `FILE` class. In this example, any object instantiated from class `CFILE` inherits the attributes from the `FILE` class.

3.3 Initialization

When defining classes, each attribute can be defined with a default value that will be assigned to objects at instantiation time. If a default value is not provided by the administrator, the system will provide its own default values. In general, it is a good practice to provide initial values in order to make sure that an object is instantiated with the right values. Initialization is also important from the consistency-model

```
# 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
```

Figure 5: Class Inheritance

```
VERSIONABLE :: superclass ENTITY;
    version_num : integer = 0;
    state       : integer = 0;
    locker      : user;
    reservation_status : (CheckedOut,Available,None) = None;
    version     : text    = ",v";
end
```

Figure 6: A class Definition with Initial Values

point of view. This will be discussed in section 3.5.3.

Figure 6 gives an example of initial values of attributes. Small attributes are assigned simply by asserting a literal value following the declaration. The *user*, *clientid* and *time* attribute can be initialized using one of the special operators mentioned above. Medium attributes are initialized by assigning a quoted string. This string represents an extension (postfix) that will be given to the file. The full name of the corresponding file on instantiation will be "object-name" concatenated with the extension. Note that the "." is not provided by MARVEL and has to be explicitly added to the default extension value. Extensions are important since some tools depend on such extensions in their processing of files. Finally, large or link attributes cannot take default values, since their values are instances themselves which are not known at definition time.

```
Rule [ parameters ]:

    characteristic function:

    property list

   { activity }

    effect1;
    effect2;
    ...
```

Figure 7: MSL syntax for a rule

3.4 Process Model: A Rule-Based Approach

This section shows how the rules, as a whole, form the notion of process programming. Because there are two aspects to this concept, it is broken up into two sections which cover the *rule* and *assistance* models, respectively.

3.4.1 Introduction

Marvel is a rule-based software development environment. The software development process is described in terms of rules that specify the behavior of the tailored Marvel environment. Ideally, any action would be modeled as a *rule*. Currently, however, built-in commands are not modeled as rules[2]. The syntax for a rule in MSL is depicted in Figure 7.

The *parameters* are the run-time objects that the rule is invoked with. The *characteristic function* section binds other objects (also referred to as *derived parameters* or *bound variables*) that the rule needs to have in the *activity* or *property list* section. The property list is a logical expression that must be satisfied for the rule to fire. The activity encapsulates the action of invoking an external tool with the specified arguments. Finally, the *effects* are a set of multiple, mutually exclusive assertions onto the objectbase that logically describe possible effects of the tool's execution. Before we show an actual example, we will first describe each of these five sections.

Rule Construction

3.4.2 Parameters

The parameters to a rule are specified by a list of VARIABLE:CLASS pairs, e.g., COMPILE [?F:CFILE]. In Marvel a variable is represented by a "?" followed by an

[2]Marvel 3.1 is expected to have the capability to overloaded built-in commands as rules.

(QUANTIFIER CLASS VARIABLE suchthat (EXPRESSION)

QUANTIFIER : forall, exists.

CLASS : A class defined in the Data Model.

VARIABLE : The unique identifier used to name this set of objects which satisfy the given expression.

EXPRESSION : A combination of AND, OR, NOT with any of the following elements.

Navigational	*Relational*
`(ANCESTOR [VAR VAR] )`	`(BVAR operator BVAR)`
`(MEMBER [BVAR VAR] )`	`(BVAR operator OP)`
`(LINKTO [BVAR VAR] )`	

Figure 8: Syntax for bindings

identifier (see Section A.1.4). When the user invokes the COMPILE rule with an object of class CFILE, MARVEL invokes its overloading mechanism (as described in section 3.4.7) to find the appropriate rule to fire.

3.4.3 Characteristic Function

The *characteristic function* section binds objects beyond the parameters, that are needed for the evaluation of the property list or for the execution of the activity. There are several reasons why a rule might want to bind other objects. The primary reason is that the firing of that rule is dependent upon the values of attributes of other objects. An example of this can be found in the BUILD rule in the C/MARVEL environment (see Appendix B). Before the rule builds a program, it checks to make sure that all the libraries that it uses have been archived by evaluating its property list. (The property list is explained in section 3.4.4.) Another reason is that the rule might need additional information to execute properly. In the BUILD rule, the activity needs to have a list of include files, which it passes to the `cc` tool (the activity section is described in detail in section 3.4.5).

The characteristic function is composed of possibly multiple *bindings* connected by AND. The AND keyword is in fact redundant and does not represent a logical AND. The syntax of an individual binding is outlined in figure 8.

The difference between `BVAR` and `VAR` in the example is that `BVAR` refers to a specific attribute of an object, whereas `VAR` refers to an entire object.

The quantifiers are mainly used in the property-list and are not part of the characteristic function, although they appear there. The only time a quantifier affects the characteristic function is when an expression returns an empty binding set and the variable is quantified with an `EXISTS` quantifier. In this case the property list is not evaluated at all and returns `FALSE`. Quantifiers will be discussed in section 3.4.4.

The `CLASS VARIABLE` pair specifies a formal parameter and the class of objects that are allowed to be bound to it. The terms *derived parameters* and *bound variables* will be used interchangeably to denote these parameters. Note that the inheritance mechanism applies here as well, i.e., any object of any subclass of `CLASS` that meets the expression will be bound as well.

The binding expression consists of nested subexpressions connected by the logical operators `AND, OR` and `NOT`.
There are two types of binding expressions: *navigational* and *associative*. The navigational type uses the *large* and *link* attributes to determine the value of the bindings, while the associative kind filters all the objects of the given class by evaluating a logical expression based upon *small* attribute values. Medium attributes are never used in this section. In fact they are only used in the *activity* section of the rule as a reference to files in the file system. There are three basic operators for the navigational binding, which we address now in turn. As a result of the binding, the VARIABLE will be bound to zero, one, or more objects. Note that this is different from a parameter symbol, which is always bound to a single object.

The *ancestor* operator allows the rule to bind a VARIABLE to an object in the composite-hierarchy that is an ancestor of a given object. For example, the following finds all ancestors of object **?x** that are of type PROJECT:

```
(exists PROJECT ?p suchthat (ancestor [?p ?x]))
```

The *member* operator is a more restricted version of ancestor in that it binds objects based upon direct parent-child relationship. For example, suppose there exists an object **?x** that has an attribute *?x.cfiles*, which is a SET_OF `CFILE`. To gather all of **?x's** children together into **?c**, the binding is as follows:

```
(forall CFILE ?c suchthat (member [ ?x.cfiles ?c ]))
```

Linkto is similar to member since it only probes one level deep, but it works on link attributes (as defined in section 3.2.2). For example, suppose there exists an object **?x** that has an attribute *?x.reference*, which is a SET_OF `link CFILE`. Here is the binding that collects all the objects of class CFILE that **?x** links to:

```
(forall CFILE ?c suchthat (linkto [ ?x.reference ?c ]))
```

All three operators can be applied to get the inverted operation when the bound variable is used as the second argument in the expression, i.e., the *ancestor* operation binds descendants, the *member* operation binds children and the *link* operation binds object that point to the first argument through a named attribute. For example, the following expression will bind all descendants to ?desc.

```
(exists CFILE ?desc suchthat (ancestor [?p ?desc]))
```

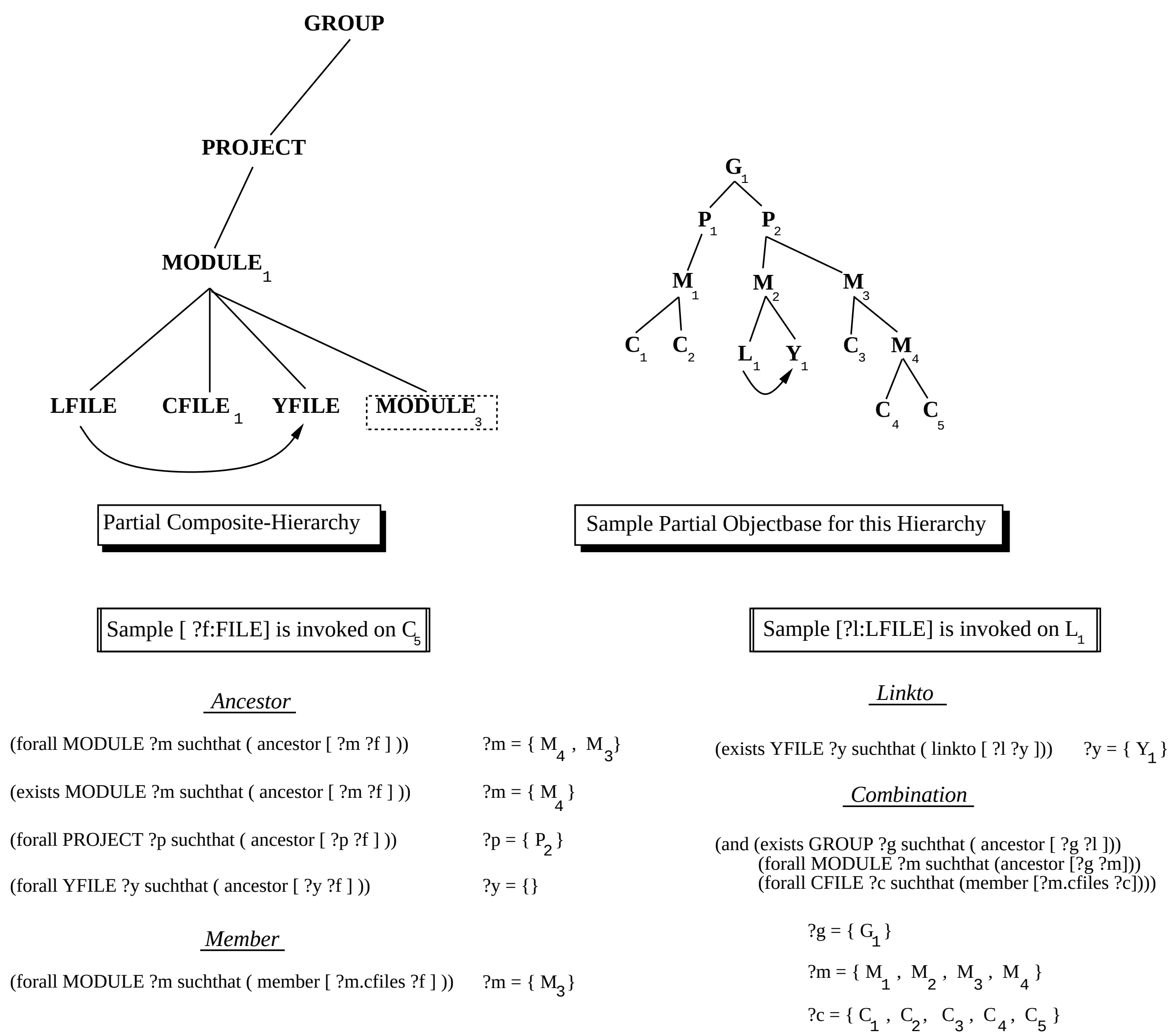


Figure 9: Instances of *ancestor* and *member* operators

Figure 9 provides more examples for the ancestor and member operators. These three operators are powerful enough to allow a rule to traverse the composite-hierarchy to bind any objects that it needs.

The associative operator allows the rule to choose objects from a specific class based upon a logical expression evaluated for each object. For example, a rule could select all the CFILE objects that have been analyzed but not yet compiled with:

```
(forall CFILE ?c suchthat (and (?c.analyze_status = TRUE
                                (?c.compile_status = FALSE)))
```

Note that this completely ignores the composite-object hierarchy. Although navigational queries can be quite expensive (e.g., when using the "descendent" operation), typically, an associative expression is more expensive since it needs to traverse all the objects for a given class and all its subclasses. It is not recommended, therefore, to use only associative query in an expression. However, it is useful sometimes to combine navigational and associative expressions with *AND* clause, to restrict the set of bound objects.

3.4.4 Property List

The property list of a rule specifies the logical state of the objectbase that must be true for the rule to fire on the given arguments. Since the *characteristic function* section allows the rule to collect an arbitrary number of objects together, the property list can be very expressive, and is not limited to the parameters of the rule. There is often confusion between the role of the characteristic function and the role of the property list. The main difference is that the characteristic function deals with *binding* objects, whereas the property-list evaluates specific properties on those objects. The output of the characteristic function is a set of objects bound to each of the symbols, and the output of the property-list is either TRUE of FALSE. Thus, even though a syntactically identical expression can appear in the characteristic function (as an associative expression) as well as in the property-list, they serve different purposes: the first will be used to filter objects that don't meet the expression whereas the second will be used as a predicate and will return a boolean value that tells whether the set of bound objects meets the condition or not.

The following example clarifies the distinction between characteristic function and property-list and how to use them properly.

Figure 10 provides the characteristic function and property list for two rules that appear similar to each other, COMPILE-1 and COMPILE-2.
However, they might behave differently on certain inputs. COMPILE-1 will bind to **?i** *only* member include files that are archived and will check that the cfiles bound to **?f** are analyzed and compiled, whereas COMPILE-2 will bind to **?i** *all* member include files and will check that they are all archived as part of the evaluation of the property-list.

```
Compile-1  [?f:CFILE]:

    (and (exists GROUP ?g suchthat (ancestor [?g ?f]))
         (forall INC ?i suchthat (and
                                     (member [?g.includes ?i])
                                     (?i.archive_status = true))))
     :
     (and (?f.analyze_status = true )
          (?f.compile_status = false))

     ...
```

```
Compile-2  [?fi:CFILE]:

    (and (exists GROUP ?gr suchthat (ancestor [?gr ?fi]))
         (forall INC ?in suchthat (member [?gr.includes ?in])))

    :
    (and (?in.archive_status = true)
         (?fi.analyze_status = true)
         (?fi.compile_status = false))

    ...
```

Figure 10: Difference between Characteristic Function and Property List

```
Compile [?f:CFILE]:

     (and (exists GROUP   ?g suchthat (ancestor [?g ?f]))
          (exists PROJECT ?p suchthat (ancestor [?p ?f]))
          (forall INC     ?i suchthat (or (member [?g.includes ?i])
                                          (member [?p.includes ?i])
                                          (linkto [?p.link_inc ?i])))
          (forall HFILE   ?h suchthat (member [?i.hfiles ?h])))

     (and (?i.archive_status = TRUE)
          (?f.analyze_status = TRUE))

     { COMPILER compile ?f.contents ?f.object_code ?h.contents ?f.error_msg "-g" }

     (?f.compile_status = TRUE);
     (?f.compile_status = FALSE);

                                                        Compile Rule
```

```
ENVELOPE
SHELL sh;
INPUT
    text          : thefile;
    binary        : obj_file;
    set_of HFILE  : ifiles;
    text          : error_msg;
    literal       : CCFLAGS;
OUTPUT
;
BEGIN
...
END     Compile Envelope
```

Figure 11: Compile rule from C/MARVEL

Thus, assuming that the properties on **?f** are TRUE, in the case where some of the member include files are not archived, COMPILE-1 will return TRUE (with a subset of include files bound to **?i**) whereas COMPILE-2 will return FALSE. Besides this logical difference in functionality, there is another important reason why the condition part of the rule has been divided into a characteristic function section and property list section: The administrator has more control over defining the *assistance model*, which applies only to the property-list section of a rule. MARVEL'S assistance model provides a way to automatically satisfy the property list of a rule if it is not currently satisfied. Thus, COMPILE-1 does not provide any means for the rule processor to affect the states of the include file objects bound to **?i**, whereas COMPILE-2 does. This is discussed in detail in section 3.5.

Quantifiers

As mentioned above, quantifiers are specified in the characteristic function but are actually applied in the evaluation of the property-list. Each bound variable has a quantifier attached to it, which determines how to evaluate a predicate in the property list. In the presence of only a single bound variable in a predicate, the evaluation is straight forward: for a universally quantified variable all bound objects must satisfy the predicate and for an existentially quantified variable at least one must satisfy the predicate. For example, in the **compile** rule in figure 11, both predicates in the

```
Compile[?c:CFILE]:
  (exists HFILE ?h suchthat (linkto [?c.reference ?h]))
   :
  (?c.time_stamp < ?h.time_stamp)

  { COMPILE ...}

  (and
    (?c.compile_status = TRUE)
    (?c.time_stamp = CurrentTime));

  (?c.compile_status = FALSE);
```

Figure 12: A Two-Variable Predicates

property list apply to **?f**, a universally quantified variable, and thus all bound objects must satisfy the predicates in order for the predicate to return TRUE.

In the presence of two variables in a predicate, the evaluation is more complex. Figure 12 shows a simple example of a predicate that involves two variables. In this example, there is a comparison between the time_stamp attributes of `?c` and `?h`. `?c` is a parameter, so we treat it as an existentially quantified. `?h` is a bound variable, so it might be bound to a set of objects. Since `?h` is existentially quantified, the property will be TRUE if at least one of the objects bound to `?h` satisfies the predicate. When both symbols are bound variables, each of them can be quantified in either way, so there are four possible ways to evaluate a two-variable predicate in a property-list. All evaluations involve the cross product of the sets of objects bound to each symbol. (For more information, refer to the implementor's manual). In addition, there is a special, fifth case, where both variables refer to the same symbol (See figure 13). In this case, instead of looking at the cross product, each element of a set is compared against itself only, since it is more natural to think of such a comparison as between different attributes of the same object.

3.4.5 Activity

The *Activity* section of a rule specifies the actual external tool to be invoked and the interface to it, i.e., the input and output arguments. MARVEL's approach to tool integration is "black-box" integration. This means that any off-the-shelf tool that is available on Unix can be integrated into MARVEL as-is, without further modifications. However, since Unix tools are file-based and MARVEL is object-based, there has to be some interface in which objects are mapped to the corresponding files and vice versa. This is done through a mechanism called an *envelope*. Section 4 explains in

```
Compile[?c:CFILE]:
   :
  (?c.source_time_stamp > ?c.object_time_stamp)

  { COMPILE ...}

  (and
    (?c.compile_status = TRUE)
    (?c.time_stamp = CurrentTime));

  (?c.compile_status = FALSE);
```

Figure 13: A Two-Variable Predicate with same symbol

detail the envelope mechanism and how to write envelopes. For now it is sufficient to describe how an envelope and its arguments are specified in MSL. The MSL syntax for the invocation is:

```
{ tool_name tool_method input-arguments RETURN output-arguments }
```

where `tool_name` is the tool's class name, `tool_method` is the specific method within the class, `input-arguments` are the arguments to be sent to the tool, `return` is a keyword that specifies that all following arguments are output arguments, and the `output-arguments` are the output from the tool back to MARVEL.

An example of a `tool_name` and a tool method was given in figure 4 in the definition of the COMPILER class: There, the `tool_name` is COMPILER, and the specific `tool_method` is COMPILE.

Input arguments can be small attributes, literals, or medium attributes. Medium attributes are in fact MARVEL's interface to the "hidden" file system, and they enable to pass files to the tool. For backward compatibility, input arguments can be also objects, which are being translated internally to a directory in the file system, but new MARVEL environments should not use objects as input parameters. Small attributes are translated into their ASCII representation and then passed to the envelope.

Figure 11 provides an example of the invocation of an envelope that compiles a `cfile` file. The characteristic function collects all the objects that the compiler needs in order to compile this file. Specifically, the rule needs the `hfiles` that the `cfile` might reference. The input arguments then are:

- *?f.contents* is the medium attribute that maps to the file(s) on the file system that contains the source code for this object(s).

```
Assign [?child:AS_0, ?SO:test4]:
  :
  (?SO.state0 = Ready)
  { ASSIGN get_user return ?uA }
  (?child.owner = ?uA);
```

Figure 14: Example of Output Arguments in a Rule

- *?f.object_code* is the medium attribute that specifies where the object code resulting from the compilation will be stored.
- *?h.contents* is the medium attribute that specifies where the source code for the include files are on the filesystem.
- *?f.error_msg* points to a file that records the compilation process and reports any errors that occurred.
- "-g" is a literal sent to the compiler to specify that the compiler should produce additional symbol table information for the debugger.

Output arguments are limited at this point to being an untyped stream of bytes, which are typed later on, when actually used[3]. As an example of such a rule, consider figure 14.
Here there are no input arguments and **?uA** is a single output argument. When the envelope returns, **?uA** will be bound to a string, which is asserted to the **?child.owner** attribute.

3.4.6 Effects

The final section of a rule is a list of mutually-exclusive effects, which Marvel can assert once the envelope's execution has completed, and not before. Since the activity is treated as a "black box", it must report back to Marvel the results of its execution. Currently, the envelope returns an integer that Marvel uses as an index to choose the proper effect to assert. That is, if the envelope returns zero, the first effect will be asserted, if it returns one the second one will be asserted, and so on. If the envelope returns a return code outside the scope of possible effects, none of the effects is asserted and an error message is printed to the user's screen. Note that the return code is different then the return arguments mentioned above. An envelope always returns a return code back to Marvel as an implicit argument. It is the responsibility of the envelope to provide a meaningful return code that will correspond to the effect to be asserted by Marvel. Conventionally, a zero return

[3]A useful extension planned for 3.1 is for the envelope to return a set of strings that represents a selection of a subset of files sent to the envelope, based on their contents.

code corresponds to a successful invocation and the various non-zero return codes correspond to various exceptions or errors in the invocation of an envelope. However, nothing in MARVEL restricts the administrator to follow this approach.

In the compile rule (figure 11), there are two effects: The first, which asserts that `(?f.compile_status = TRUE)`, is used when the compilation process is successful, and the second, `(?f.compile_status = FALSE)` is used in the case of failure. Since the envelope can modify only files, (thus indireclty it modifies medium attributes) while MARVEL modifies the small and large attributes, great care must be taken to ensure that the envelope and rule coordinate their actions, so that the contents of the file system and the objectbase will be consistent with each other.

Assertions can be made only on *small* attributes and on *link* attributes. Link attributes can be used with either `linkto` or `unlink` operators. Figure 15 gives an example of rules that assert link attributes. In the first rule, the *?root* object is being linked to three parameter objects in an effect. In the second rule, the characteristic function locates and binds linked objects, and the effect unlinks them from the object bound to *?root*. Note that while the first rule's assertions are only on parameters there will be a single link operation per effect, whereas the second rule's assertions involve bound variables, and therefore might imply zero, one, or more `unlink` operations per effect.

An important restriction in the effects part of the rule is that it is not allowed to make assertions on bound variables, except in link/unlink assertions. The main reason for this is due to the assistance model (see section 3.5) and due to the current concurrency-control mechanism in MARVEL. There is, however, a workaround to get the effect of modifying bound variable using chaining. It will be explained in section 3.5.

3.4.7 Rule Selection

MARVEL allows for multiple rules with the same name to co-exist. Two kinds of rules with the same name can co-exist in the system: Rules that have identical number, types, and order of parameters, and rules that differ in one or more of the above. When a rule is invoked with actual parameter objects, MARVEL applies an overloading procedure that selects the most appropriate rule(s) for execution. Once this set of rules is determined, MARVEL's rule processor fires the first of these rules whose condition is satisfied.

Rule Overloading

Like multi-methods, rules are identified through the unique combination of their name and the types, order and number of their parameters. MARVEL performs a variation of Breadth-First-Search (BFS) on the class hierarchy to find the "closest" rule. For each rule, a vector of BFS numbers that corresponds to the number of parameters is generated. The vector represents, for each object, the distance, in BFS order (left to right at same level), between the type (class) of the actual object parameter and the type of the corresponding formal parameter of the rule. If there is no ancestral

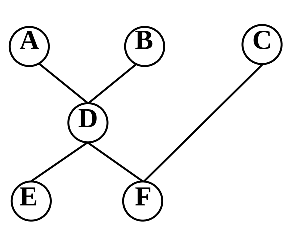
```
# link first three parameters to ?root, which is an ancestor of the
# fourth parameter.

Bind [?a:DOCFILE, ?b:DOCFILE, ?doc:DOCUMENT, ?SO:AS_3]:
  (exists ROOT_CLASS ?root suchthat (ancestor [?root ?SO]))
  :
  { }
  (and (linkto [?root.a ?a])
       (linkto [?root.b ?b])
       (linkto [?root.doc ?doc]));

# bind in the char. function all DOCUMENT and DOCFILE objects linked to
# ?root (an ancestor of ?SO) via the doc, a, and b attributes,
# and unlink them from ?root.

Unbind [?SO:AS_3]:
  (and (exists DOCUMENT ?doc suchthat (linkto [?root.doc ?doc]))
       (exists DOCFILE ?b suchthat (linkto [?root.b ?b]))
       (exists DOCFILE ?a suchthat (linkto [?root.a ?a]))
       (exists ROOT_CLASS ?root suchthat (ancestor [?root ?SO])))
  :
  { }
  (and (unlink [?root.a ?a])
       (unlink [?root.b ?b])
       (unlink [?root.doc ?doc]));
```

Figure 15: Assertion on `link` attributes




```
CLASS A :: superclass ENTITY;        1. RULE  R [ ?a:A, ?b:B ];
CLASS B :: superclass ENTITY;        2. RULE  R [ ?a:A, ?d:D ];
CLASS C :: superclass ENTITY;        3. RULE  R [ ?c:C, ?d:D ];
CLASS D::  superclass A, B;          4. RULE  R [ ?d:D, ?c:C ];
CLASS E::  superclass D;             5. RULE  R [ ?c:C ];
CLASS F::  superclass D, C;
```

Figure 16: Inheritance and Polymorphism of Rules

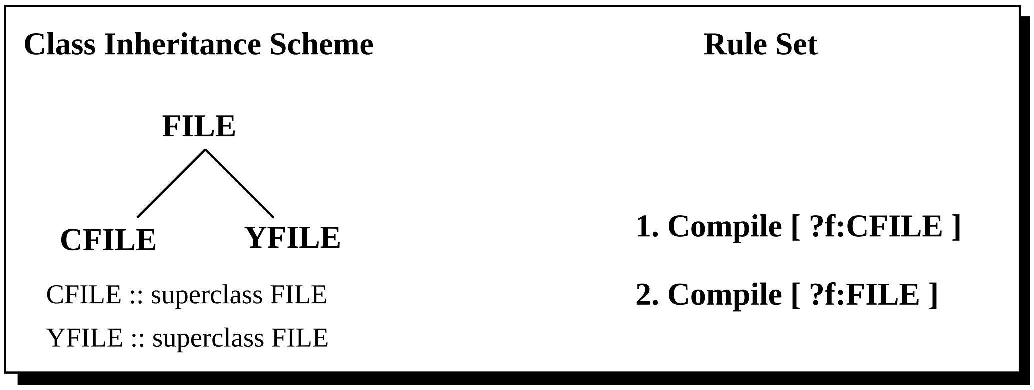


Figure 17: Rule overloading example

relationship between a formal type and an actual type, or if the number of actual and formal parameters is different, the rule is disregarded. Then, the rule with the "minimal" vector, i.e., the rule whose formal parameters are closest in BFS order to the types of the actual parameters entered by the user, is selected. For example, assume there are six classes `A,B,C,D,E` and `F` and five rules, all of which have the name `R` but whose formal parameters are of different types, as shown in figure 16. If a user requests to fire a rule with two parameters that are instances of class `E` and `F`, respectively, MARVEL will produce the following vectors for the five rules: [2,3], [2,1], disregard, [1,1], and disregard. Hence, the fourth rule is the one that will be selected for execution. Note that the third rule was disregarded because there is no ancestral relationship between class `E`, the first actual parameter's type, and class `C`, the rule's first formal parameter's type.
In another, simpler example (figure 17), If the COMPILE rule is invoked with an object O_c of class CFILE, then the first COMPILE rule will be selected. If, however, the COMPILE rule is invoked with an object O_y of class YFILE, MARVEL chooses the second rule, since YFILE is a subclass of FILE and the second rule operates on objects of class FILE. Note that if there were a third COMPILE rule with a parameter of class YFILE, then COMPILE$[O_y]$ would activate this third rule since it is more specific than the second rule. If the second rule didn't exist and the user requested the COMPILE rule on an object O_f of class FILE, MARVEL would report that the COMPILE rule can't be invoked with that argument.

For more information refer to [11].

3.4.8 Importing and Exporting Strategies

Rules are written in files called *strategies*. Each MSL source file must have a ".load" extension since the loader searches only for such files. MARVEL provides for some degree of modularity in writing rules, using the `import` and `export` operators. An MSL file can *import* arbitrarily many other MSL files, by listing them after the `import` keyword. During translation, import is like an Ada with clause rather than a C

include. That is, classes and tools defined in an imported MSL file may be used in the importing file; however, rules defined in an imported MSL file are not considered in any way by the translator. During loading, import is treated like an include, in that all imported files (and the files they import, etc.) are automatically taken by the loader and converted to MARVEL 's internal representation.
It is not meaningful to say anything except `all` following the `exports` keyword. True export restrictions have not been implemented in MARVEL.

3.5 Assistance Model

The assistance model is intended to perform two main tasks: To enact the process by means of automatic invocation of activities and to maintain the consistency of the objectbase by means of propagation. Note that the assistance model does NOT help in the *definition* of the consistency of the project, but only in preserving consistency. Both of these tasks are carried out through a *chaining* engine managed by the rule processor. We first describe the general chaining mechanism as applied for automation, and then talk specifically about consistency maintenance.

Chaining

A *chain* in MARVEL is a logical connection between two rules, specified by a *match* of a predicate in the *effect(s)* of one rule with a predicate in the *property list* of another. Figure 18 shows one such example: The `(?f.analyze_status = TRUE)` predicate in the effect of the ANALYZE rule satisfies the predicate in the property list of the COMPILE rule. MARVEL maintains a static *rule network* for all the rules loaded in the system, which specify these connections for all the predicates of all the loaded rules. It can be modified by the administrator to operate in special modes (will be explained in section 3.5.4) or when a new set of rules is loaded dynamically. (See section 5 for more information.)

When a rule is invoked by a user via the client process (as explained in section 1.3), the rule processor first checks whether the condition is met on that rule. If the condition is not met, MARVEL attempts to satisfy it by *backward chaining.* If the condition is already satisfied or backward chaining succeeded in satisfying the condition, the rule's activity is being sent to the client for execution. When the activity ends, it returns to the rule processor all the necessary information, and the rule processor asserts the right effect(s). After the assertion, the rule processor checks whether other rule's conditions are met as a result of the recent assertion. If there are such rules, *forward chaining* is applied to fire these rules.

Conceptually, backward chaining can be viewed as a process of "repairing" the state of the world, by recognizing a situation and enabling a desired activity to fire, whereas forward chaining can be viewed as a process of "improving" the state of the world by triggering other activities. We now explain backward and forward chaining in detail.

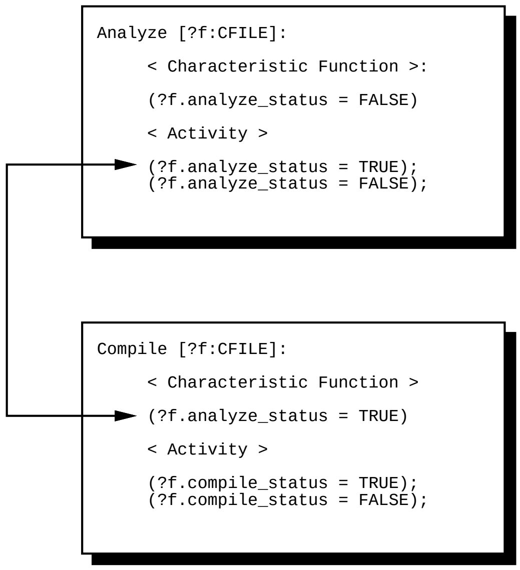


Figure 18: A connection between predicates

3.5.1 Backward Chaining

Backward chaining attempts to satisfy the property list of a rule that has been invoked. Currently backward chaining is initiated only when a user requests a rule to fire. When a rule is invoked, and the property list is not met by the objects bound to the rule, it is because one of the predicates in the property list failed. This predicate is named the *failed_predicate* and the object that caused it to fail is the *failed_obj*. The rule processor then searches the rule network for a rule that asserts this particular predicate, and thus can potentially make the failed attribute true and satisfy the condition of the user-invoked rule. Figure 19 provides an example of backward chaining; C-2 is an object of class CFILE whose *C-2.analyze_status* is FALSE.

If the COMPILE rule from figure 11 is invoked with C-2 as its argument, the property list is not satisfied. Since the rule network has a connection between the ANALYZE rule and the COMPILE rule (see figure 18), MARVEL attempts to satisfy the failed predicate by backward chaining to the ANALYZE rule with C-2 as its argument. If the ANALYZE activity decides that the object was successfully analyzed, then the first effect is asserted, that *C-2.analyze_status* = TRUE. Note that this is the state of the objectbase necessary for the original COMPILE[C-2] invocation to fire. In this case, we say that the backward chaining *succeeded* in satisfying the property list of the COMPILE rule. On the other hand, if the ANALYZE rule asserted the second effect, that *C-2.analyze_status* = FALSE, then the state of the objectbase has not changed, hence COMPILE [C-2] is unable to fire. In this case, we say that the backward chaining *failed* to satisfy the property list of the COMPILE rule.

The backward chaining algorithm follows the AND-OR tree mechanism of typical backward chaining systems, with one significant exception. Assume that r_j has been in-

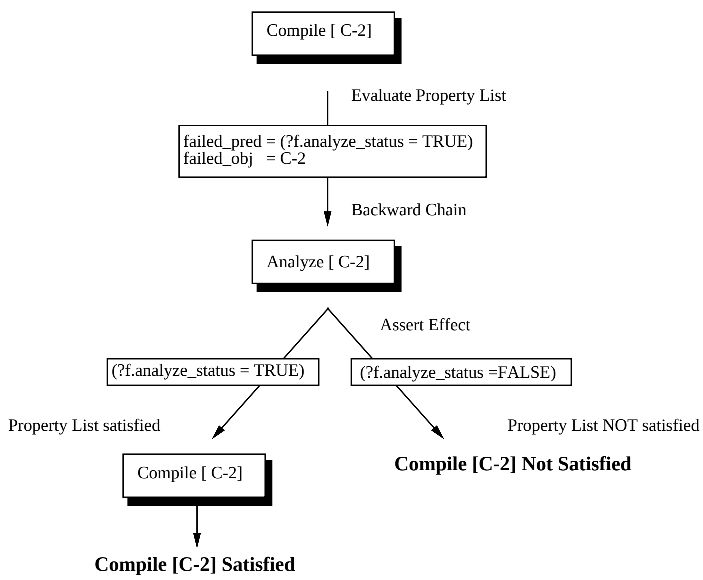


Figure 19: Backward Chaining to Satisfy a Property List

voked, and the system attempts to satisfy it by firing rule r_i, if possible. Since rules may have multiple alternative effects, it is possible for the property list of a rule r_i to be true, but firing the rule does not produce the required effect for the backward chaining on rule r_j to succeed; this is the exact situation outlined above when the second effect is asserted. In other words, failure to satisfy a condition can happen either because the condition of the backward chained rule is not met, or because the effect asserted after firing the rule was different than the one that actually satisfies the condition of the original rule. The algorithm will repeatedly try all possible rules that could satisfy a failed predicate until the predicate is satisfied, or the list of candidate rules is exhausted. As in other backward chaining systems, it is a recursive procedure, i.e., if the condition of the rule that is applied by previous invocation of backward chaining is itself not satisfied, backward chaining is applied to it in order to satisfy it.

3.5.2 Forward Chaining

Forward chaining can be viewed as an opportunistic approach. Once assertions of a rule take place, the rule processor looks for all opportunities to fire other rules whose conditions became satisfied. As in backward chaining, forward chaining is a recursive process, i.e., a forward-chained rule that was fired triggers other rules and so on. However, only rules whose condition is fully satisfied can be fired. In other words, Marvel does not provide for backward chaining while forward chaining in order to satisfy a condition of a rule. Another important characteristic of forward chaining is that it is *event-driven*. That is, rules will forward chain only on objects that are directly affected by the previous assertion. Section 3.5.8 explains how parameter objects are selected for rules during chaining.

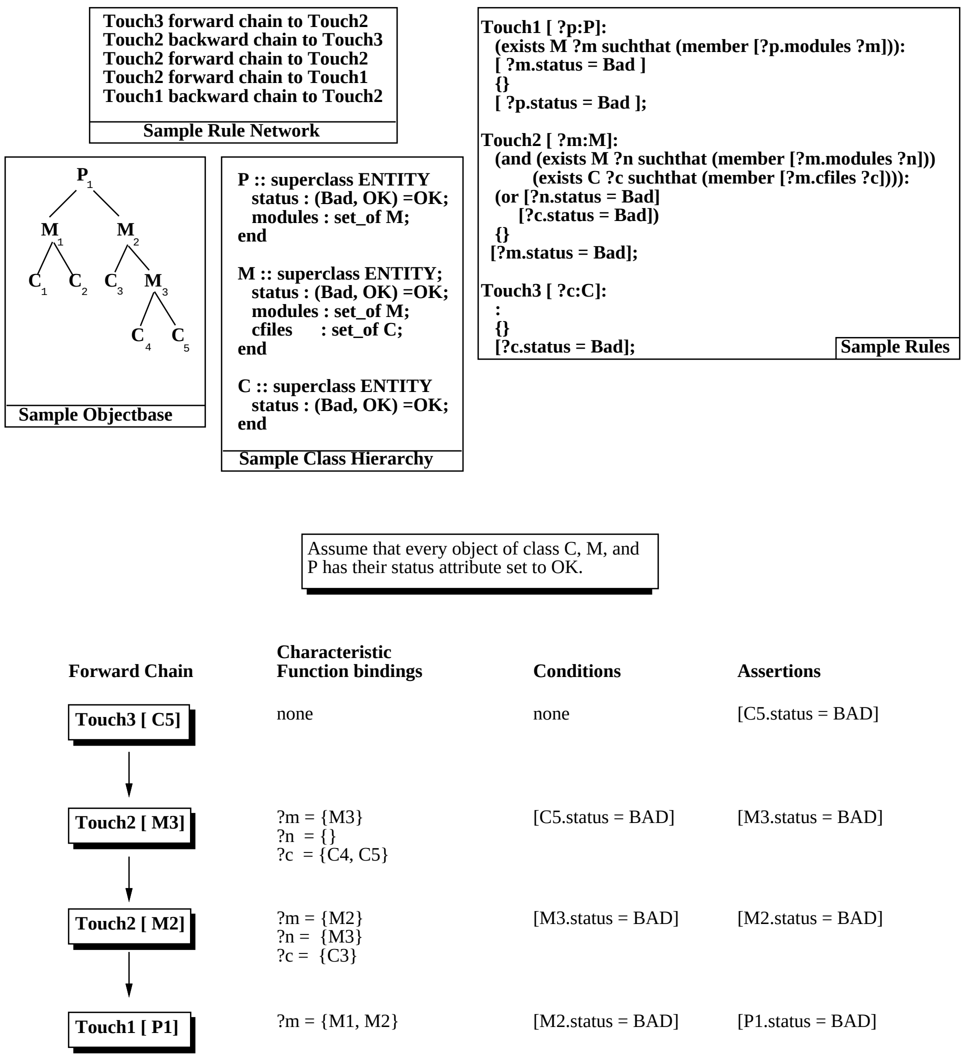


Figure 20: An example of a forward rule chain

Figure 20 provides a detailed example of a typical forward chain. As stated in the figure, it is assumed that every object in the given sample objectbase has its *status* attribute set to OK. When the user invokes TOUCH3 on **C5**, there are no conditions to satisfy, so the rule is executed (it has no activity) and asserts a change on the objectbase, that *C5.status* = OK. This triggers the forward chain to TOUCH2 on **M3**. It evaluates its characteristic function and creates the sets for ?m, ?n, and ?c. When the property list is evaluated, it is found to be true since *C5.status* = BAD. This rule executes its null activity and asserts *M3.status* = BAD on the objectbase. This triggers the forward chain to TOUCH2 on **M2**. After its characteristic function is evaluated, its property list is found to be satisfied since *M3.status* = BAD. This triggers the chain to TOUCH1 on **P1**, whose property list is satisfied by *M3.status* = BAD, and it asserts its effect on the objectbase, completing the chain.

3.5.3 Consistency Maintenance

Up to now the chaining mechanism was explained in the context of automation. We now explain how it used for consistency maintenance. The consistency of the objectbase is defined by the combination of initial values as specified in the class definition and by a special kind of predicates in the rules, called *consistency predicates*. The role of chaining is then to ensure that the consistency of the project is preserved according to its definition. Consistency predicates are identical to automation predicates except that they are enclosed with "[]" brackets. Just like automation predicates, they can appear either in a property-list or in the effects of a rule. The main difference between these kinds of predicates is that automation predicates imply "best-effort" request to enact the process, that is, if a chained rule cannot be invoked for some reason (e.g., conflict with another user), then the chain will be stopped. On the other hand, a consistency predicate implies that the propagation must take place or else the chain must be rolled back. In other words, any chained rule must be fired (recursively) and either all or none of the propagations take place.

Thus, we can distinguish between two types of chains, namely automation and consistency chains. Consistency chains are characterized by having a consistency predicate as the source of each edge of the chain. Thus, a link between a consistency predicate to an automation predicate is still considered a consistency link. All other chains are considered automation chains. This means that one chain can consist of consistency sub-chains and automation sub-chains. For purposes of concurrency control, consistency implications have higher precedence over automation implications.

A problem arises when a chain that involved some activities has to be rolled back, since some of the activities cannot be reversed. (e.g., sending mail, printing a document, etc.). For that matter we distinguish between *inference rules* and *activation rules* . Inference rules are rules with empty activities and activation rules do have activities. Consistency chains then, are restricted to consist of inference rules. Another property of inference rules is that they do not have to go to the client for execution, and therefore the entire chain is executed atomically at the server. (If you need to refresh your memory with the architecture and its terminology refer to section 1. For more information on the implementation of the architecture and the concurrency control see the implementor's manual.)

Finally, consistency chains happen only in forward chaining. The reason is that assuming the project is consistent when initiated, and assuming the atomicity property of consistency chains, there is never a reason why a consistency backward chain should take place. The implication is that any consistency predicate in the property list is assumed to be true for all objects in the objectbase that can be accessed by the predicate at the time the rule is fired. If this is not the case for some reason, (e.g., the initial value of an object does not match the requirement in the predicate), there is a problem with the consistency model. We don't have currently ways to detect "wrong" definitions of consistency.

For example, if the user modifies an HFILE, the CFILES that access this HFILE should

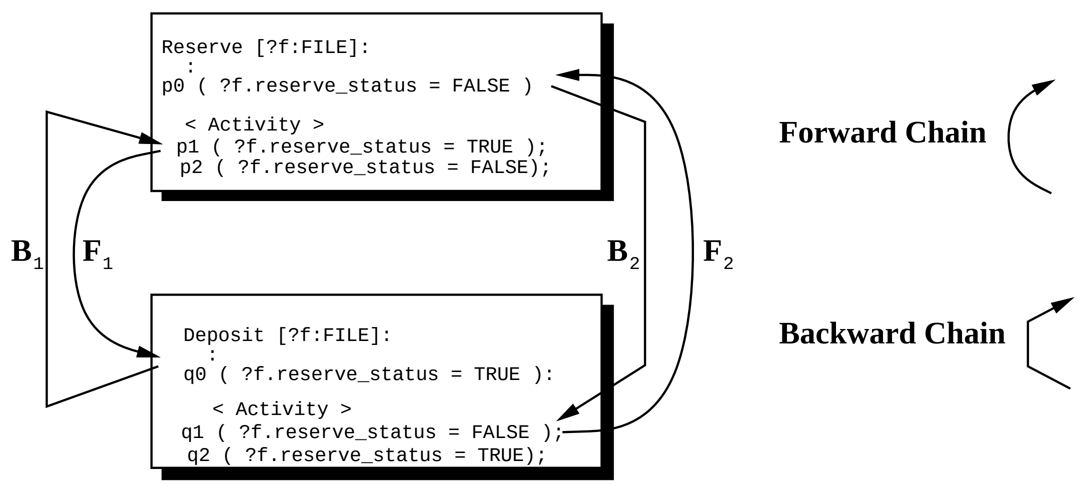


Figure 21: Default Rule Network Generation

be marked as being not analyzed. The primary difference between these methods is transaction recovery. If an automation forward chain aborts, there is no specific need to undo the effects of the forward chain. On the other hand, if a consistency forward chain aborts, MARVEL should reverse the effects of the rule chain to maintain the objectbase in a consistent state. Transaction recovery is outside of the scope of this manual, but it should be noted that it has some support in the MSL language.

3.5.4 Chain Control

When a set of rules is loaded into the MARVEL kernel, MARVEL creates the *rule network* where nodes represent rules and edges represent chains among rules. Recall that a chain between two rules is determined by a match between a predicate in one rule and other predicate in other rule. (Self edges are not allowed). However, by simply matching predicates in such way, the edges are not directed, i.e., an edge between two rules represent a chain in both directions. This network might result in a behavior that the project administrator does not want. A clear example is seen in figure 21.

Without any constraints, these two rules will generate two forward chains and two backward chains, denoted by F_i and B_i respectively. For example, the RESERVE rule has a forward chain F_1 (from predicate $p1$) to the DEPOSIT rule (to predicate $q0$). This means that RESERVE might forward chain to DEPOSIT, meaning that it is not reserved anymore. This is clearly not desired. Furthermore, this can easily lead to an infinite-loop, if the same effects are asserted over and over. (Notice that in general MARVEL will not prevent the creation of cycles in the network, since cycles in the network do not represent the actual runtime invocation sequence due to the multiple effects property.) In general, we want to be able to "direct" the graph, so that edges in one direction will not necessarily imply the existence of edges in the opposite direction. Furthermore, we would like to be able to "trim" the graph as we

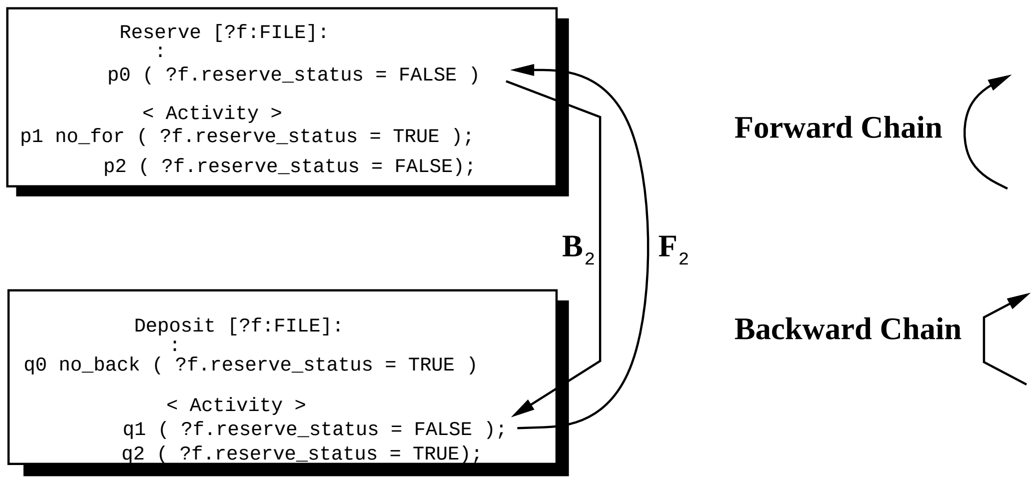


Figure 22: Control Over Rule Network Generation

wish by eliminating any outgoing or incoming chain (edge) on a predicate basis.

For this matter, MARVEL provides chaining *directives* to control the chaining network. The available directives are **no_backward**, **no_forward**, and **no_chain**, and any of them can be applied to predicates in the effect or in the condition . The **no_backward** directive on an effect prevents any condition predicate to backward chain into it, and when applied on a condition it prevents the rule to backward chain to another rule. Similarly, **no_forward**, when applied to an effect predicate, means that no forward links are generated from the predicate, and when applied to a condition predicate it means that no rule can forward chain into it. Finally, the **no_chain** directive prevents any kind of chain from or into the predicate. Thus, we get full control over chaining. Another notation for **no_chain** is `(* .. *)`

Figure 22 shows a possible modification of RESERVE and DEPOSIT rules. Here, $p1$ is augmented by the **no_forward** directive, which prevents MARVEL from forward chaining from the RESERVE rule to the DEPOSIT rule. In addition, $q0$ predicate of the DEPOSIT rule is augmented with a **no_backward** directive, which will prevent it from backward chaining to reserve. B_2 and F_2 are left, with the semantics that a file cannot be reserved if it is already reserved without depositing it first, and one cannot deposit anything unless it was reserved first. If this behavior is not desired it can be easily modified by adding directives.

Chaining Modes

In addition to the static and local specifications, the administrator can toggle between two modes of operation with respect to chaining:

1. Full - Chaining as specified. This is the default mode.
2. No_Automation - Disable automation chains.

For more information on invoking this command refer to the users manual.

Note that once a mode is selected, it affects all users of the environment, that is, chaining mode is global and not per-user parameter.

3.5.5 Chaining Approach - A Methodology

The task of chaining is to enact a process as defined by the rules and the data model. The problem is how to write the rules in order to define the process so that chaining will reflect the intended purposes of the process writer. The simplest approach is to treat each rule in isolation, i.e., specify for each rule the conditions and effects as implied by the activity of the rule and let the rule processor infer the necessary chains. This approach is used in forward chain systems (e.g., expert systems) in which the main task of the system is to make the inferences implied by the facts and rules and conclude a (new) result. However, it is usually not suitable for Marvel for the following reasons:

1. Marvel is using rules in order to enact a specified process model. In order to have a coherent process model, the administrator has to look at chains that he explicitly wants to have and for chains that he wants to eliminate as part of his process. Furthermore, chains can be used as a flow-of-control mechanism in which each rule in a chain is a "subroutine" that performs a specific task on a specific set of arguments. As an example, using chains one can have the effect of asserting values on a bound variable through chaining to another rule that receives those variables as parameters.

2. The fact that Marvel uses both backward and forward chaining complicates the processing of rules and thus may lead to unexpected and not desired results if one assumes the writing of each rule in isolation. An example is the chains created in the reserve-deposit rules. Furthermore, the multiple mutually-exclusive effects complicate even more the chaining possibilities. Thus, it is very easy to lose control over the process.

3. The consistency model necessarily requires looking at a set of rules and how they interact since it implies propagation of effects on objects throughout the objectbase, which entails chaining of rules to assert those values. Note that unlike automation, if consistency predicates are not specified correctly they will impose an effective consistency model that is different then the one the administrator had in mind.

Another approach would be to write all rules with full awareness of chaining, i.e., each predicate is written in the context of how and where it will chain. The problem with this approach is that as the rule base grows, it becomes harder to predict all possible chains.

We found out that the best way to program a process using MARVEL rules is by a hybrid approach:

1. Define the data model, with the process model in mind.
2. Define a set of rules with some chains in mind. In particular, design chains that describe automation activities that you want to chain, and eliminate some other chains using the predicate directives.
3. Define carefully consistency chains in order to make the required propagations. This implies writing new rules and possibly modifying rules already defined in step 2. Note that inference rules are usually used to propagate consistency. section 3.5.6 gives a way to hide those rules from the end-user.
4. Load the set of rules into MARVEL using the **load** command, and examine the rule network using the chaining-graph utility (see the user's manual).
5. Build a prototype objectbase or use the MARVELIZER tool (section 5) to migrate an existing directory into MARVEL's data model. Now you can test your rules on the data and see if they perform properly.
6. If you have to change the rules but not the data-model, simply change the rules and goto step 4. If the data model needs to be changed, remove the prototype objectbase, reload, and rebuild a new objectbase, and continue testing. Note that currently there is no utility in MARVEL to evolve the schema.
7. Once you are satisfied with the behavior, build the "real" objectbase and allow clients to use the environment.

3.5.6 Hiding Rules

An administrator can prefix a rule with the keyword `hide` in order to tell MARVEL that this rule should not be visible to the end-user and is used internally, usually by the consistency-propagation rules. The idea here is to hide these rules from the user, both to prevent him from invoking them, and to avoid confusion with respect to these rules, which usually do not map to activities.

3.5.7 How to drive the process

Since MARVEL has both *forward* and *backward* chaining, the rule writer has several options for determining how to chain rules together to drive a certain task. A specific task can be backward-chain driven, forward-chain driven or both. One guideline to follow is the *focus* of the task. If it is well defined, and there is a set of well-understood goals (and sub-goals), then backward chaining should be used to enact the control. On the other hand, forward chaining is used to propagate changes through the objectbase.

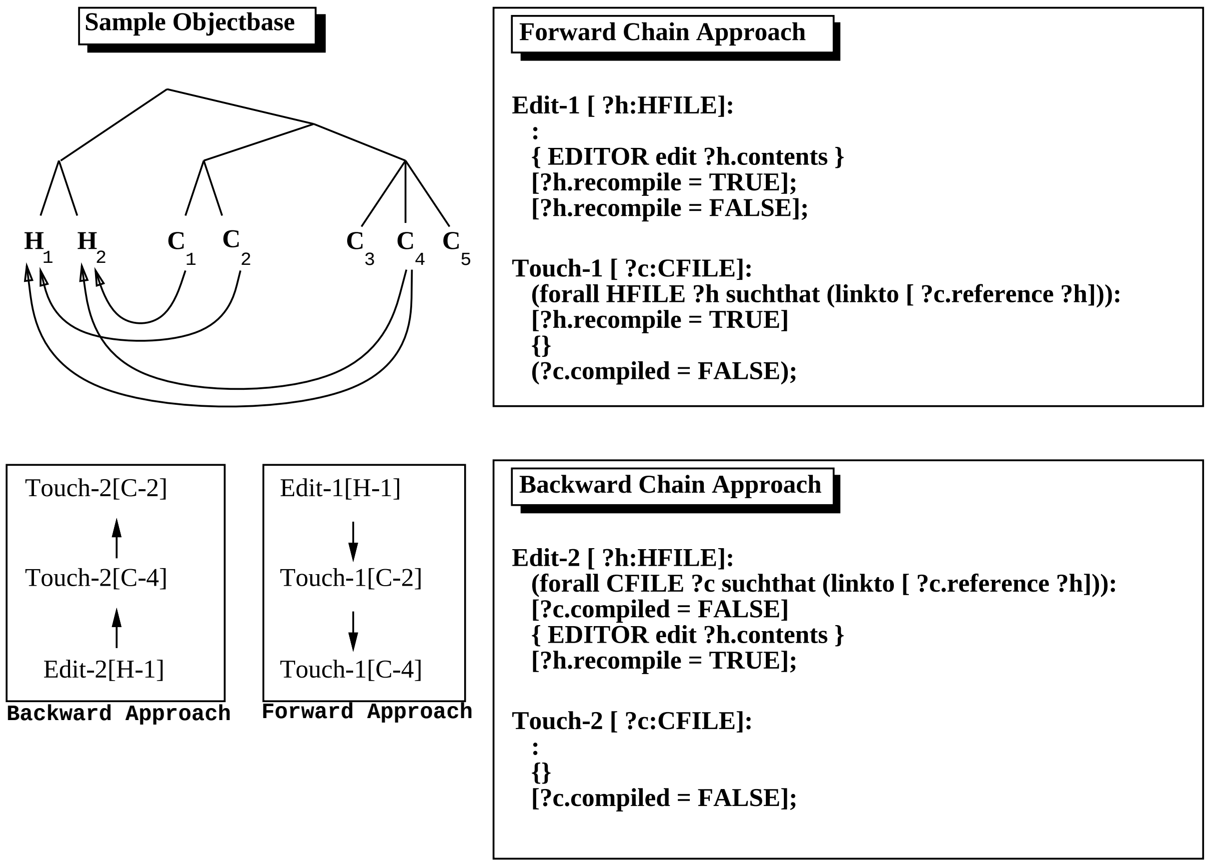


Figure 23: Chaining approaches

Figure 23 gives two examples that enact the same task by backward and forward chaining respectively. Given an HFILE that is included by CFILE objects as a header file, we want to mark the CFILES as `NotCompiled` when the HFILE is edited. We assume that there are *links* from the CFILES to the HFILE through a `link` attribute.

The EDIT-1 rule of the forward chain approach edits an HFILE and marks it for recompilation by asserting the *?h.recompile* attribute to be TRUE. Then, during the forward chaining cycle, MARVEL chains off of the `[?h.recompile = TRUE]` predicate to the TOUCH-1 rule. This rule is invoked for all the CFILE objects that link to this HFILE.

The EDIT-2 rule of the backward chain approach has a constraint that requires all the CFILE objects that refer to this HFILE to be first marked so that their *?c.compiled* attribute is set to FALSE. MARVEL backward chains to the TOUCH-2 rule for each of the CFILES, then executes the EDIT-2 rule.

```
Edit[ ?h:HFILE]:
  :
 { EDITOR edit ?h.contents }
  (?h.recompile = TRUE);



Compile [?c:CFILE]:
   (exists HFILE ?h suchthat (linkto [?c.reference ?h]))
    :
    (?h.recompile = TRUE)
    { COMPILER compile ?c.contents ?h.contents }
     (?c.compiled_status = TRUE);
     (?c.compiled_status = FALSE);
```

Figure 24: Inversion Example

3.5.8 Passing Arguments Between Rules

When chaining between rules occurs, the rule processor has to determine the proper parameters of the chained rules. Since chaining is predicate-based, when the target predicate is on a bound variable (defined in the characteristic function of the rule), it is not clear what objects to choose as parameters.

Recall from section 3.4.3 that the characteristic function of a rule has a set of known objects (the parameters) and generates a set of *bound variables* based upon navigational and associative queries to the objectbase. This is a well-defined process that, given the same objectbase state, produces the same set of bound variables for the same input. If we attempt to reverse this procedure, we find the results are not as well defined. Specifically, if we are in the process of chaining, and are chaining to a predicate which is based on a bound variable, we have to determine the proper arguments to use for this chained-to rule. This situation is clearly seen in figure 23 when the user invokes TOUCH-2 with C-4 as an argument. This asserts *C-4.compiled* = FALSE on the objectbase, which causes a forward chain to EDIT-2 based on the predicate in its property list. However, we must still determine the parameter ?h that this rule will use. By inspection, we see that either H-1 or H-2 will satisfy the property list of EDIT-2 (we assume the administrator wishes to have this behavior. If that is not the case, then the administrator must write the predicates so as to control the chaining, as in section 3.5.4).

The Algorithm

The intuition behind this approach is that each navigational-query is itself invertible (not as in mathematics, but as a shorthand for reverse-evaluate). For example, in figure 24, we want to establish the constraint that if an `HFILE` object is edited, all `CFILES` that link to it must be (re)compiled.

The COMPILE rule's characteristic function
`(exists HFILE ?h suchthat (linkto [?c.reference ?h]))`
returns all `HFILEs` that are linked-to by `?c` through the `reference` attribute. In this case, `?c` is bound to an object (the parameter of the rule) and `?h` is bound to a set of object after evaluation.

In the case of a chain from EDIT to COMPILE through the predicate:
`(?h.recompile = TRUE)`
We have `?h` bound to an object (passed from the EDIT rule) but we don't have the parameter `?c` bound. However, as seen in section 3.4.3, the evaluation process can be inverted to bind to `?c` all `CFILE` objects that link to `?h` through the `reference` attribute.

The general algorithm follows:

1. start from the predicate that caused chaining. This predicate has an object bound to a symbol in the predicate, passed to it by the chained rule.
   If the symbol happens to be the parameter symbol, we are done. The parameter object is found.

2. "Climb-up" the characteristic function and invert the evaluation to get a newly bound symbol. Each time, check if the new symbol is the parameter symbol. If it is, the parameter is found.

Several notes about this algorithm:

- At any point in the algorithm, an evaluation might result in a set of objects bound to a symbol. This means that each object has to be considered in the evaluation of the next step, resulting in sets of objects bound to the various symbols. The outcome is that multiple instantiations of a rule, (each with different parameter object) will be chained for execution, if multiple objects are bound to the parameter symbols.

- The algorithm works only for single-parameter rules. Passing arguments between multi-parameter rules is not implemented yet. This means that you cannot plan on chaining into a rule with multiple parameters, although you can chain from a rule with arbitrary number of parameters, as long as the destination rule has a single parameter.

- If all queries in the characteristic function are navigational, the process is guaranteed to succeed. If a characteristic function has a clause which is purely associative, the algorithm might not find the parameter object. This is another reason why purely associative queries are not encouraged in MARVEL . However, a combination of associative and navigational queries in an `AND` clause, will still allow to find the parameter objects always.

For more information on parameter-passing refer to [9].

3.6 The Environment Directory

All the information about a specific environment is contained in a specific directory, called the "environment" directory.
When either the server or the client are invoked, they try to connect to the current directory as the environment. Alternatively, one can specify the full path of the environment directory as an argument to the client or the server. If the server or client think it is an invalid MARVEL environment, it will refuse to connect and will exit. The way MARVEL identifies a directory as being a MARVEL environment, is by looking for the existence of a special file called .marvel_id. Note, however, that if the daemon is running and the client didn't specify a directory and its current directory is not valid the daemon will not be able to start the server, since it not being run interactively. See the user's manual for instructions on how to invoke the server and the client.

The actual contents of the environment directory consists of the following types of files/directories that reside on a typical MARVEL environment:

- MSL source files, denoted by a ".load" (mandatory) extension
- Envelopes - there are two types of envelopes: source envelopes, written in SEL , and target envelopes, generated by SEL . The latter type of files have an ".env" extension. the source files have no special extensions.
- `data` directory - This directory is maintained by the MARVEL server. It contains an `objectbase` file, which is a binary file accessed through gdbm, and the most recent backup copy of it named `objectbase.old`. In case of a disaster that might have corrupted your objectbase, you might want to replace your objectbase with the backup copy of it. Another important file in this directory is the `strategy` file. It is an ASCII file that contains an intermediate representation of the data, process model, and the chain network. The loader keeps a copy of that file in the directory. You should not modify either the `strategy` or the `objectbase` files manually as this can lead to an undefined results (and a likely crash of MARVEL when invoked on this directory).
- hidden file system - all the binary and text attributes that represent files are stored there along with the file system hierarchy that represents the objectbase hierarchy. Each top-level root object in the objectbase will have a directory named after it in here.
- system files and directories for use by MARVEL are :
  - .client_id_ctr - a file that maintains a persistent counter of the clientid to ensure that clientids are not recycled.
  - .server_port - This file is there only when a server is running. It should not be there if no server is being run on that directory. If you are absolutely

sure that no server is running on the directory and the file is still there remove it, otherwise no server will be bale to run on this directory.

- tx_log - this directory is used by the recovery manager to recover in case of a transaction has to be rolled back.
- `Marvel-client.xxxxx.log` - these are log files generated by the client. the "x" stands for a unique number that prevents from multiple logs from different clients to end up in the same file. They should be inspected upon unexpected crash of either the server or the client.
- `Marvel-loader.log` - This is log generated by the loader process.
- `Marvel-server.log` - This is log generated by the server. Of all three log types, it has the most important information. This log file is appended, so records of previous invocations are available for inspection.

4 SEL programmer's guide

4.0.1 Envelopes

The tool invocation process occurs in the envelope. We have extended the standard UNIX shell scripts to support typed passing of parameters and returning multiple typed values. The envelope writer is required to explicitly declare all the attributes that it is receiving. Not only does this allow for type-checking, but it provides a very clean interface between MARVEL and the envelopes. For example, observe the following envelope which compiles a given CFILE (recall the activity invocation from figure 11):

```
ENVELOPE
SHELL sh;
INPUT
          text      : thefile;
          binary    : obj_file;
   set_of INC       : ifiles;
          text      : error_msg;
          literal   : CCFLAGS;
OUTPUT none;
BEGIN

tmp_dir=/tmp/compile$$
mkdir $tmp_dir

# we need to make the -I list
idir=""
if [ "x$ifiles" != "x" ]
then
    ln -s $ifiles $tmp_dir
    idir="-I$tmp_dir"
fi

cc $CCFLAGS -c $idir $thefile -o $obj_file -ll -lc -lm -lX11
cc_status=$?

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

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

```
    echo compile successful
    RETURN "0";
else
    echo compile failed
    RETURN "1";
fi

END
```

An envelope consists of an SEL framework, specifying the input and output parameters of the envelope and the shell to be used for interpreting it, surrounding the shell script that will actually be executed.

The framework begins with a line containing the keyword "ENVELOPE" and an optional name for the envelope. If a name is provided, it must be followed by a semicolon. Next comes a line specifying which shell is to be used to run the script; the choices available here are sh, ksh, and csh. After this preamble comes the parameter specification. The input parameters come first, followed by the output parameters. Each parameter declaration contains, the type and name of the parameter, separated by a colon. Note that the TYPE appears FIRST, contrary to the way that declarations are specified in MSL. Each declaration is terminated by a semicolon. Both the input and output sections are required; if one is to be empty, the keyword "none" should be used. After the output section is the shell script, between BEGIN...END delimiters.

The shell script in an envelope has four conceptual parts. First, the envelope takes the objects passed to it and performs some initialization procedure. (In this case, a temporary directory is created in /tmp which contains soft links to all of the included HFILE that this CFILE needs. The second part executes the tool(s). The third part "cleans up" after running the tool. The fourth and final part collects and returns the envelope outputs and status code. In this case, only a status code is being returned. Note that the status code returned to MARVEL is an integer value greater than 0, and must match against an effect in the *effects* section to be asserted.

Envelopes should use the RETURN statement, rather than the shell's own exit command, to return their status code and any output parameters that the envelope may have. In the RETURN statement, the return status must be written as a string; the double quotes are required. The make_envelope program will not complain about scripts that use exit instead of RETURN, but such scripts will produce incorrect behavior[4].

As an example of an envelope that returns something other than a status code, consider the *Assign* envelope from section 14. For convenience, we reproduce this envelope below. The envelope takes no input parameters, but produces a single string-valued output parameter in addition to its return code. The user name entered is

[4]For backward compatibiltity, SEL currently supports "old-format" scripts that are simply shell programs, and do use exit instead of RETURN, but new scripts should be written in SEL

returned as the output string, while the return code is 0 if a non-null string was entered as the user name and 1 if a null string was entered.

```
ENVELOPE Assign;

SHELL sh;

INPUT
   none;

OUTPUT
    string : ret_string;

BEGIN

echo "Enter userid to be assigned:"
read ret_string

if [ "x$ret_string" = "x" ]
then
    echo "Must specify a user id"
    RETURN "1" : "" ;
fi

RETURN "0" : $ret_string;

END
```

This envelope is much simpler than the Compile envelope. It has no initialization section, and the tool section simply prompts for and reads a string from the terminal. Since there was no initialization, there is also no clean-up, and the output section checks for a null input string and returns an appropriate status code, also returning the input string in the successful case.

The declaration of an output parameter looks exactly like that of an input parameter, except that it appears in the OUTPUT rather than in the INPUT section. Due to implementation restrictions, currently only single-valued output parameters are supported; set-valued output parameters should be supported in a future version of Marvel. Basically, any value that can be represented as a single string of characters, containing no spaces or special characters, can be returned as an output parameter.

In the script, output parameters may be used and set in the usual manner for shell variables, with the restriction that they will not have an initial value and so should be set before they are used.

If there are multiple output parameters, they should be separated by commas in the RETURN statement. All output parameters are required, that is, a value must be provided for each output parameter in every return statement in the envelope. This serves the purpose of reminding the SEL programmer to provide some value for each output parameter in every case, avoiding any possible problems that could be caused by not having a value set at a later point, after the values have been returned to Marvel.

While the value provided to the RETURN statement will usually be the variable declared for the parameter in the OUTPUT section, as in the normal return in the example envelope, it may also be a literal string or any other shell variable. The error return in the example envelope returns a literal null string as the ret_string parameter.

For more imformation on the envelope mechanism, refer to [10].

5 Administrator's Built-in commands

This section describes the subset of the Marvel commands that can be invoked only by the administrator. For the rest of the commands, refer to Marvel user's manual. For actual help on invocation of a command, refer to the user's guide, or the on-line **help** or **usage** commands. In order to run as an administrator, use the -a option when calling Marvel , and make sure that your userid is in the `administrators` system file (section 3.6).

5.1 Load: Loading an MSL Environment into Marvel

The `load` command is used to load a set of MSL files that represent an environment into Marvel. This command invokes a separate program, called simply "loader", which translates the MSL specifications into an intermediate representation that is loaded to the server. After an environment is loaded once, there is no need to re-load it on subsequent invocations of the server, unless the administrator wants to modify the environment. In this case he invokes the `load` command again with the new set of strategies. The `load` command replaces the entire existing environment with the new one, i.e., it ignores the previous environment. Thus, there is no way to replace or merge a subset of the strategies and leave the others intact, although the same effect can be achieved by loading the unchanged strategies along with the new ones.

The current interface to the loader requires to have one main file that imports all the other MSL files. The translated files are kept in an intermediate form in the data directory of the environment, for subsequent invocations of the server.

The loader can also be invoked externally to Marvel . This might be sometimes useful since it runs faster outside of Marvel . However, you have to be familiar with the usage of it. Typing `loader` with no arguments will print the calling sequence.

5.1.1 Usage of Load

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

Enter strategy name:

You respond with the name of the main strategy of the environment. Do not specify the .load extension. The tutorial in the user's manual has an example of loading a strategy.

5.2 Reset

The `reset` command is used to fix inconsistencies in an objectbase, mainly in the debugging phase of the environment. It allows the administrator to change any attribute of any object to any acceptable value, at any time. **reset** is not intended to be used normally, and is a dangerous command since it interferes with the process model. `reset` performs type conversion followed by type checking to verify that the new value to be assigned has the appropriate type.

When the user clicks on `reset`, the following line appears on the screen:
`Enter parameters < class object > attribute < value >:`
Where:

- `class` and `object` denote the object to be modified and its class. If omitted, the current object is assumed.
- `attribute` is a mandatory parameter that denotes the name of the attribute to be changed.
- `value` is the value to be given to the attribute. If omitted, the default value is assigned.

5.3 CRload

The `crload` command should not be used.

5.4 Shutdown

This command is used to shut down a server, if it was started manually. If the server was started by the daemon there is no need to use the command since the server shuts itself down when no users are logged in. The invocation is straight forward. The server will wait sixty seconds before actually exiting. Since there is no asynchronous communication between the server and the client, clients are not notified unless they actually interact with the server within that period.

5.5 MARVELIZER : Immigration of software into MARVEL

This section contains a description of the MARVELIZER , an immigration tool for constructing or augmenting MARVEL objectbases to represent existing software artifacts copied from the data repository of some source SDE. The immigration process depends heavily on the source SDE's structure. MARVEL 2.6 had two immigration tools available: The MARVELIZER, for simple immigration, and the Complex Marvelizer (CM), for immigration of complex source SDEs. However, MARVEL 3.0 has only a simple MARVELIZER . More information about the complex MARVELIZER can be found in [13].

The MARVELIZER is intended for immigration of data repositories that are characterized by the fact that they don't have a database that holds state or connective information and a DBMS to manipulate it, or alternatively, it is sufficient to migrate the data repository without interaction with the database management utilities. In the very simple case, the data repository can be a flat, unorganized group of files, all within the same directory, where control and management of the files is placed upon the user. The data repository can also be in the form of a file system directory structure that implies relations among directories and the files that they contain, where a data item is represented as a file or as a directory containing component items (files). Finally, it can have, in addition to the file system structure, specially formatted files and directories maintained by tools that are treated as part of the SDE. Thus, this form includes "private" data repositories of individual tools, such as the delta files of RCS or the Unix `sccs` tool, whose contents are intended to be hidden from users. Notice that in this case, the tools that manipulate the "private" data must also be copied or converted into Marvel.

We'll describe the practical steps a user would take to prepare to use MARVELIZER, and then describe the process itself by an example.

Preparation Steps

1. Prior to Marvelizing (immigrating), the MARVEL administrator must create a MARVEL data model to define the structure of the destination object base. The data model could be defined with either of two goals in mind. The MARVEL class lattice could be defined to mimic all or most of the structure of the source SDE; this of course makes Marvelization relatively easy, but would be done only when a new MARVEL environment was being developed specifically to take over the role of the source SDE. The alternative is for a MARVEL administrator to develop a data model suitable for the purposes of the new MARVEL environment, independent of whether or not the environment is planned to encompass existing data items from some other SDE that will later be Marvelized. In all our examples, here and in following sections, the data model was developed prior to the MARVELIZER tool, and thus was conceived entirely independently of the formats of any potential source SDEs.

2. Write File Conversion (FC) specifications for all the files in the original system. FCs provide information about the kinds of files that can be encountered in the source data repository, and how these map to the kinds of data items represented in MARVEL. These specifications specify filename patterns, generally suffixes (e.g., filename extensions) of all the different source file types that map to the

given destination classes. Patterns may also match entire filenames, but do not involve the contents of files.

```
<d-class> <s-f-pattern-1> ... <s-f-pattern-n>
```

`<d-class>` refers to a destination class in MARVEL, and `<s-f-pattern>` refers to a filename or pattern matching files in the source data repository.

3. Write Directory Conversion (DC) specifications for all directories that might be automatically converted. DCs show how directories in the source data repository map to particular classes in MARVEL. It is also possible to map suffixes or prefixes of directory names, as with files.

   ```
   <d-class> <s-d-pattern-1> ... <s-d-pattern-n>
   ```

   `<s-d-pattern>` refers to a directory name or pattern matching directories in the source data repository. In verbose mode, the user will be queried for any unspecified directories. These specifications are consulted before the immigration of each data item.

4. Choose either verbose or automatic modes to proceed, in general, automatic mode is used if the user doing the Marvelization does not want to monitor the process.

MARVELIZER algorithm

Immigration is accomplished via two simultaneous preorder traversals, one over the source data repository's filesystem directory structure, and the other over the MARVEL object base, starting at the destination object specified by the user. In the following description, there are notions of "current" object and class. The current object is the one being examined at some particular instant in the traversal of the destination MARVEL object base; the current class is that object's class.

When a file in the source data repository traversal is encountered, MARVELIZER checks whether the file's suffix matches a specification in the table, and if that specification's class matches either the current class (the class of the current object in the MARVEL object base), or a set attribute of the current class. In the first case the file is simply copied into the appropriate place in MARVEL's hidden filesystem space, as determined by the current object. It copies the file under a text attribute, and if there are multiple text attributes it picks the first one. In the second case, which gets priority in case both cases are true, a child object is hierarchically added to the current object; then the (source) file is copied to a place in the hidden filesystem determined by the new child object. Files not specified by a specification are skipped. MARVELIZER generates messages specifying those files that were skipped, with the explanatory content of these messages depending upon the verbosity mode chosen.

When a directory in the source data repository is encountered, MARVELIZER first looks to see if there is a matching specification. If so, and if the specification's class is the current class of the destination object base traversal, then a corresponding new object is added to the destination object base, as described above. Otherwise, MARVELIZER determines the set of possible classes this directory could be an instantiation of, based on the attributes of the current object in the (destination) object base traversal. If there is more than one matching attribute type or class, the user is queried (possibly skipping the directory is an option). This is the only direct interaction with the user once the process has started, and can be turned off, to only generate messages for the user to look at later.

This process continues recursively, until the traversal of one or the other data repository is complete. If the MARVEL objectbase traversal completes first, those remaining portions of the filesystem traversal must not match MARVEL's current data model, and must be separately Marvelized. If MARVEL's graphics interface is being employed, the visual display of its object base is updated after Marvelization. At this point, any software artifacts that were not successfully immigrated (i.e., were skipped) can be reMarvelized individually, by running MARVELIZER again using a different user-designated object as the starting point and a subset of the source SDE's filesystem as the source root. Such failures happen when MARVELIZER cannot recognize the structure of an existing source directory hierarchy, for example, when insufficient specifications were provided.

Example

We present here an example of immigrating an SDE into MARVEL using C/MARVEL's data model as described in appendix B.1. The initial dialog with the MARVELIZER for the above Marvelization is shown in figure 5.5; user responses are in *italics*. The steps are:

1. Specify the location of the original SDE in the filesystem.
2. Pick the root object for Marvelization, i.e., where the marvelized tree will link-to as sub-tree in MARVEL . This option is available only in the graphical user interface, where objects can be clicked. In the command-line interface marvelizations must start at the top-level.
3. type the attribute to connect the marvelized information to the root-object (Again, only when a root object exists).
4. Optionally, one can save the state of the objectbase before marvelization.
5. Enter mode of operation.
6. Enter FCs
7. Enter DCs

```
Enter the root directory of the information to be Marvelized:
/example/SDE
Pick the parent object for Marvelization, or <cr> for top-level:
user picks an object, e.g "src"
Enter attribute to connect marvelized information to ``src'':
modules
Do you wish to record the current objectbase?  [Y/N]:
Y
(v)erbose, (q)uiet or (s)ilent mode?  (any other key to exit):
v
Now enter all file suffixes for each class.
Format is:
CLASS_NAME <suffix-1> <suffix-2> ...  <suffix-n>
Enter a q when finished, or an e to exit.
Enter string: FILE .c .o Makefile
Enter string: VERSION ,v
Enter string: q
Now enter specific directories and the classes to immigrate them to.
Format is:
CLASS_NAME <directory-1> <directory-3> ...  <directory-n>
Enter a q when finished, or an e to exit.
Enter string: MODULE sort search
Enter string: q
Ready to Marvelize /example/SDE. Are you sure [y/n]: y
```

Figure 25: A dialog with Marvelizer

A MSL Reference Manual

This is the full definition of the MSL language. It is written in the forms of tokens (terminals) and productions. The parser and semantic analyzer are implemented in yacc, an LALR(1) shift-reduce parser generator, and the lexical analyzer is implemented in lex, a lexical-analysis generator, which recognizes regular expressions. Familiarity with yacc and lex will help to understand this section but is not required. Familiarity with context-free grammars is required.

A.1 The Tokens

A.1.1 Basic patterns

```
DIGIT       [0-9]
LETTER      [a-zA-Z]
BOTH        {LETTER}|{DIGIT}
LETTERS     {LETTER}+
SPACES      [ \t]
IDSTRING    {LETTER}({LETTERS}|{DIGIT}+|\_)*
SUFFIX      ({BOTH}|\,|\.)*
COMMENT     \#.*
QUOTEID     "[0-9a-zA-Z]*"
QUOTESTR    "\"([^\"]|\\\")*\"
COMMENT     "#.*
```

A.1.2 Keywords

These keywords are reserved.

CurrentClient	clientid	imports	no_forward	strategy
CurrentTime	consistency	insert	not	string
ResetClient	end	integer	objectbase	suchthat
ancestor	end_objectbase	link	or	superclass
and	exists	linkto	real	text
automation	exports	member	remove	time
binary	false	nil	return	true
boolean	forall	no_backward	rules	unlink
built_in_overload	hide	no_chain	set_of	user

A.1.3 Special Tokens

```
( ) { } [ ]
```

```
=    -- EQ_OP_tok   (EQ)
<>   -- EQ_OP_tok   (NEQ)
<=   -- EXP_OP_tok  (LEQ)
>=   -- EXP_OP_tok  (GEQ)
>    -- EXP_OP_tok  (GT)
<    -- EXP_OP_tok  (LT)
::   -- D_COLON

+=   -- MATH_OP_tok (PLUS_EQ)
-=   -- MATH_OP_tok (MINUS_EQ)

(*   -- OTHER_LEFT_KW  (NO_CHAIN)
*)   -- OTHER_RIGHT_KW (NO_CHAIN)
```

A.1.4 Numbers and Identifiers

This section outlines some of the very low-level details that lex understands. It uses the standard notations of regular expressions. A character in quotes is a literal character, | represents options, + indicates one or more of the specified item, * represents 0 or more of the specified item. The ? represents an optional item. Items in (...) are groupings, while items in { ...} indicate a user-defined character class (see section A.1.1).

```
"-"{DIGIT}+ | {DIGIT}+                         -- IVAL

"-"?{DIGIT}*"."({DIGIT}+) |
"-"?{DIGIT}*"."({DIGIT}+)(E|e)"-"?{DIGIT}+     -- RVAL

"?"{IDSTRING}                                  -- VARIABLE
"?"{IDSTRING}"."{IDSTRING}                     -- BVAR
"?"{IDSTRING}":"{IDSTRING}                     -- PARAM

{IDSTRING}("/"{IDSTRING})+                     -- PATH
{IDSTRING}                                     -- ID
{QUOTEID}                                      -- QUOTE_ID
{QUOTESTR}                                     -- QUOTE_STR
"{SUFFIX}"                                     -- FILE_NAME
```

A.2 The Productions

This section outlines all the productions of the grammar. Multiple entries denote alternative derivations of a non-terminal.

start	STRATEGY_KW ID imp_exp objbase rule_section over_section
imp_exp	IMPORTS_KW imp_name_list ; EXPORTS_KW exp_name_list ;
imp_name_list	**nothing**
	ID
	imp_name_list , ID
exp_name_list	**nothing**
	ID
	exp_name_list , ID
objbase	**nothing**
	OBJECTBASE_KW classes ENDOBJBASE_KW
classes	class
	classes class
class	ID D_COLON superclasses attributes END_KW
superclasses	SUPERCLASS_KW ;
	SUPERCLASS_KW super_name_list ;
super_name_list	ID
	super_name_list , ID
attributes	attrib
	attributes attrib
attrib	ID : noninitable_type ;
	ID : autoinitable_type ;
	ID : initable_type ;
	ID : initable_type EQ_OP_TOK init_val ;
autoinitable_type	USER_KW
	TIME_KW
	CLIENTID_KW
initable_type	STRING_KW
	INTEGER_KW
	REAL_KW
	BOOLEAN_KW
	file_type
	enumerated_type
noninitable_type	ID
	SETOF_KW ID
	LINK_KW ID
	SETOF_KW LINK_KW ID
enumerated_type	(et_name_list)
file_type	TEXT_KW
	BINARY_KW
et_name_list	ID
	et_name_list , ID
init_val	ID
	FILE_NAME
	PATH
	QUOTE_STR_KW

	QUOTE_ID_KW
	BOOL_VAL_TOK
	IVAL
	RVAL
rule_section	**nothing**
	RULES_KW rules
over_section	**nothing**
	OVERLOAD_KW rules
rules	rule
	rules rule
rule	ID [parameters] : bindings : precond activity mult_posts
	HIDE_KW ID [parameters] : bindings : precond activity mult_posts
parameters	**nothing**
	PARAM
	parameters , PARAM
bindings	**nothing**
	binding
	(BOOL_OP_TOK binding_list binding)
binding	(QUANTIFIER_TOK ID VARIABLE SUCHTHAT_KW binding_cond)
binding_expr_list	binding_cond
	binding_expr_list binding_cond
binding_cond	(set_expr)
	(expression)
	(multiple_bind_cond)
multiple_bind_cond	BOOL_OP_TOK binding_expr_list binding_cond
	NOT_TOK binding_cond
binding_list	binding
	binding_list binding
activity	{ }
	{ action }
action	ID ID act_var_list
outputs	**nothing**
	RETURN_KW OUT_VAR_LIST
out_var_item	VARIABLE
out_var_list	**nothing**
OUT_VAR_LIST OUT_VAR_ITEM	act_var_item
	QUOTE_ID
	QUOTE_STR_ID
	VARIABLE
act_var_list	**nothing**
	act_var_list act_var_item
mult_posts	;
	mult_post_list
mult_post_list	post ;
	mult_post_list post ;

post	allowed_post_cond
	(BOOL_OP_TOK post_list allowed_post_cond)
post_list	allowed_post_cond
	post_list allowed_post_cond
allowed_post_cond	consistency_cond
	automation_cond
	both_cond
	other_cond_post
precond	**nothing**
	which_cond
which_cond	allowed_pre_cond
	(multiple_cond)
allowed_pre_cond	both_cond
	automation_cond
	other_cond_pre
	consistency_cond
allowed_list	which_cond
	allowed_list which_cond
expr_cond	BVAR MATH_OP_TOK operand
operand	BVAR
	QUOTE_ID
	QUOTE_STR
	RVAL
	IVAL
consistency_cond	[solo_cond]
	CONSISTENCY_KW (solo_cond)
automation_cond	(solo_cond)
	AUTOMATION_KW (solo_cond)
other_cond_pre	OTHER_LEFT_KW solo_cond OTHER_RIGHT_KW
	NO_BACKWARD_KW (solo_cond)
other_cond_post	OTHER_LEFT_KW solo_post OTHER_RIGHT_KW
	NO_FORWARD_KW (solo_post)
	NO_BACKWARD_KW (solo_post)
	NO_BACKWARD_KW [solo_post]
	NO_CHAIN_KW (solo_post)
solo_cond	expression
	expr_cond
solo_post	POST_EXPR
POST_LINK_EXPR	
POST_UNLINK_EXPR	
multiple_cond	BOOL_OP_TOK allowed_list which_cond
	NOT_KW which_cond
expression	BVAR exp_op expression_tail
	BVAR EQ_OP_TOK BOOL_VAL_TOK
	BVAR exp_op IVAL

	BVAR exp_op RVAL
post_expr	BVAR EQ_OP_TOK BOOL_VAL_TOK

BVAR EQ_OP_TOK EXPRESSION_TAIL

expression_tail	BVAR
	ID
	QUOTE_ID
	QUOTE_STR
set_expr	MEMBER_KW [BVAR VARIABLE]
	ANCESTOR_KW [VARIABLE VARIABLE]
	LINK_TO_KW [BVAR VARIABLE]
	LINK_TO_KW [BVAR NIL_TOK]
exp_op	EQ_OP_TOK
	EXP_OP_TOK

B C/Marvel : An example environment

This appendix provides the source code for CMarvel, the environment that is used to maintain and run Marvel itself. Although it has Marvel specific support, it can be customized to any project based on its specific needs. Note that there is a simpler example environment in the user's guide. While the latter is more for learning purposes, this environment is much more elaborate, and can be viewed as a "real" example of a process and data models.

Section B.1 provides listings of MSL data-model definition, section B.2 provides listings of all MSL rules in the system that define the process and consistency models, and section B.3 provides listings of all SEL envelopes called from the various rules.

B.1 C/Marvel Data Model

```
#
#                    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.


#  The composition hierarchy looks like this : (links are not shown)
#
#                                        GROUP
#                                          |
#                            +-----------------+---------------+
#                            |                                 |
#                         PROJECT                             TEAM
#                            |                                 |
#                 ----------------------                       |
#                /       /  |  \       \                  PROGRAMMER
#               /       /   |   \       \                      |
```

```
#            BIN    DOC  INC  LIB     SRC                   |
#            /\      |     |   |       |                    |
#    SHELLFILE  EXE   |  HFILE  AFILE  MOD              MINIPROJECT
#                    |               / \
#                    |          (MOD)  CFILE
#                  DFILE
#                    |
#                    |
#                POSTSCRIPT
#
#
#


# 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


BUILT  :: superclass ENTITY;
    build_status   : (Built,NotBuilt,INC_NotBuilt) = NotBuilt;
end

HISTORY  :: superclass ENTITY;
    history : text;
end

ARCHIVABLE :: superclass ENTITY;
    archive_status : (Archived, NotArchived, INC_NotArchived) = NotArchived;
end

REFERENCED :: superclass ENTITY;
    reference : set_of link HFILE;
end

TIMESTAMPED :: superclass ENTITY;
    time_stamp : time;
end
```

```
PRINTABLE :: superclass ENTITY;
    print_request : boolean = false;
end


VERSIONABLE :: superclass ENTITY;
    version_num : integer = 0;
    state       : integer = 0;
    locker      : user;
    reservation_status : (CheckedOut,Available,None) = None;

    version     : text    = ",v";
end



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

GROUP :: superclass ENTITY;
    projects : set_of PROJECT;
    team     : set_of TEAM;
end

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

PROJECT :: superclass BUILT, ENTITY;
    status    : integer = 0;
    bin       : set_of BIN;
    doc       : set_of DOC;
    include   : set_of INC;
    lib       : set_of LIB;
    src       : set_of SRC;
    link_inc  : set_of link INC;
    link_lib  : set_of link LIB;
end


EMPLOYEE  :: superclass ENTITY;
     first_name    : string;
```

```
     last_name     : string;
end


PROGRAMMER :: superclass EMPLOYEE;
   mail_id : string;
   local   : set_of MINIPROJECT;
   server  : link EXEFILE;
   client  : link EXEFILE;
   loader  : link EXEFILE;
end


TEAM :: superclass ENTITY;
    programmer  : set_of PROGRAMMER;
    manager     : link EMPLOYEE;
    role1       : link EMPLOYEE;
    role2       : link EMPLOYEE;
    role3       : link EMPLOYEE;
end



# local area for the programmer to do work.  This contains hfiles and
# cfiles.

MINIPROJECT :: superclass BUILT, ENTITY;
    hfiles         : set_of HFILE;
    cfiles         : set_of CFILE;
    yfiles         : set_of YFILE;
    lfiles         : set_of LFILE;
    exec           : EXEFILE;
    link_afile     : link AFILE;
end


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

INC :: superclass ARCHIVABLE, ENTITY;
    hfiles : set_of HFILE;
end


BIN :: superclass BUILT, TIMESTAMPED, ENTITY;
  created_by      : user;
```

```
  configuration   : text = ".config";
  binexecs        : set_of EXEFILE;
  shexecs         : set_of SHELLFILE;
end


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

LIB :: superclass TIMESTAMPED, ARCHIVABLE, ENTITY;
    afiles : set_of AFILE;
end

AFILE :: superclass TIMESTAMPED, ARCHIVABLE, HISTORY, ENTITY;
    afile : binary = ".a";
    tags  : text   = ".TAGS";
end

SRC :: superclass ARCHIVABLE, ENTITY;
   modules : set_of MODULE;
end


# Module is the organizing force in the data model.  It groups together
# lex files, yacc files, and C files.  NOTE:  This definition is recursive,
# that is, modules can contain other modules.
MODULE :: superclass ARCHIVABLE, ENTITY;
    lfiles   : set_of LFILE;
    cfiles   : set_of CFILE;
    yfiles   : set_of YFILE;
    doc      : set_of DOC;
    modules  : set_of MODULE;
    afiles   : set_of link AFILE;
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 TIMESTAMPED, VERSIONABLE, HISTORY, ENTITY;
    owner    : user;
    contents : text;
end
```

```
# Needed for compilable type files. For example CFILE needs an object
# file.  The time stamp represents when this object was inserted into
# its respective library.

COMPILABLE:: superclass ENTITY;
    object_code       : binary = ".o";
    object_time_stamp : time;
end


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

CFILE :: superclass COMPILABLE, REFERENCED, FILE;
    status         : (New, NotAnalyzed, ErrorAnalyze, Analyzed,
      NotCompiled, ErrorCompile, Compiled,
      NotArchived, ErrorArchived, Archived) = New;
    contents       : text = ".c";
end


# Separate info for YACC files, so they get Yacc'ed
#

YFILE :: superclass COMPILABLE, REFERENCED, FILE;
   status         : (New, NotCompiled, ErrorCompile, Compiled,
     NotArchived, ErrorArchived, Archived) = New;
   contents       : text = ".y";
   ytabh          : text = "y.tab.h";
end



# Separate info for Lex files, so they get Lex'ed
#

LFILE :: superclass COMPILABLE, REFERENCED, FILE;
   status         : (New, NotCompiled, ErrorCompile, Compiled,
     NotArchived, ErrorArchived, Archived) = New;
   contents       : text = ".l";
   ytabh          : link YFILE;
end
```

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

HFILE :: superclass COMPILABLE, REFERENCED, FILE;
    recompile_mod : boolean = false;
    contents : text = ".h";
end


LOADFILE :: superclass FILE;
   status : (Active, Inactive) = Inactive;
   contents : text = ".load";
end


EXEFILE :: superclass BUILT, FILE;
    exec    : binary;
    afiles  : set_of link AFILE;
end

SHELLFILE :: superclass FILE;
    shtype : (csh, ksh, sh) = sh;
    shell  : text;
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 its own.

DOC :: superclass ENTITY;
    docfiles : set_of DOCFILE;
#   main     : link DOCFILE;
end

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

DOCFILE :: superclass FILE;
    reformat : boolean = false;
    postscript : set_of POSTSCRIPT;
    contents : text = ".tex";
    output : binary = ".dvi";
end
```

```
POSTSCRIPT :: superclass FILE;
   reformat : boolean = false;
   contents : text;
end


end_objectbase
```

B.2 C/Marvel Rules

```
#----------------------------------------------------------------------
#
#                    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
# archives a module.


strategy archive

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

imports data_model;
exports all;

objectbase

ARCHIVER :: superclass TOOL;
    stuff        : string = stuff;
    list_archive : string = list_archive;
    randomize    : string = randomize;
    update       : string = update;
end
```

```
end_objectbase

rules

# This rule archives a set of hfiles.  Why is this necessary?  Well, since
# lint calls cpp, which bombs when its input is too long, we have to create
# soft links to the hfiles to a special place so that this works
# appropriately.  Please forgive this HACK, but it is necessary.

 arch[?I:INC]:
   (forall HFILE ?h suchthat (member [?I.hfiles ?h]))
   :
   (?h.recompile_mod = false)

# eventually ARCHIVER create_link ?I ?h
   { }

   (?I.archive_status = Archived);


arch[?s:SRC]:
    (forall MODULE ?m suchthat (member [?s.modules ?m])):
    (?m.archive_status = Archived)
    {}
    (?s.archive_status = Archived);

# This rule archives a module if all its CFILEs have been archived.

arch[?m:MODULE]:
    (and (forall CFILE ?f suchthat (member [?m.cfiles ?f]))
 (forall YFILE ?y suchthat (member [?m.yfiles ?y]))
 (forall LFILE ?x suchthat (member [?m.lfiles ?x]))
 (forall MODULE ?child suchthat (member [?m.modules ?child])))
    :
    (and
         (?f.status = Archived)    # back chain to arch CFILE
         (?y.status = Archived)    # back chain to arch YFILE
         (?x.status = Archived)    # back chain to arch XFILE
         (?child.archive_status = Archived)) # back chain to arch MODULE
    {}
    (?m.archive_status = Archived);


arch[?l:LIB]:
 (forall AFILE ?a suchthat (member [?l.afiles ?a])):
```

```
 (?a.archive_status = Archived)
 { }
 (?l.archive_status = Archived);

# This archives a library (in a more virtual way) when all the modules that
# it owns have themselves been "Archived".  Note, this will gather together
# all the .o files which it owns and create the ".a" file.

arch[?a:AFILE]:
  (and (exists LIB     ?l suchthat (member [?l.afiles ?a]))
       (exists PROJECT ?p suchthat (member [?p.lib ?l]))
       (forall SRC     ?s suchthat (member [?p.src ?s]))
       (forall MODULE  ?m suchthat (and (ancestor [?s ?m])
                (linkto [?m.afiles ?a])))):

    (?m.archive_status = Archived)      # backward chain to arch MODULE

    { ARCHIVER randomize ?a.afile ?a.history}

    (and (?a.archive_status = Archived)
 (?a.time_stamp = CurrentTime));


# Archive a particular FILE (CFILE, YFILE, LFILE) into the libraries which
# any of its ancestor modules points to.

arch[?f:CFILE]:
   (and (exists MODULE  ?m    suchthat (ancestor [?m ?f]))
        (forall AFILE   ?a    suchthat (linkto [?m.afiles ?a])))
   :
   (?f.status = Compiled)        # back chain to compile

   { ARCHIVER update ?a.afile ?a.history ?f.object_code }

   (?f.status = Archived);

arch[?f:YFILE]:
   (and (exists MODULE  ?m    suchthat (ancestor [?m ?f]))
        (forall AFILE   ?a    suchthat (linkto [?m.afiles ?a])))
   :
   (?f.status = Compiled)        # back chain to compile

   { ARCHIVER update ?a.afile ?a.history ?f.object_code }

   (?f.status = Archived);
```

```
arch[?f:LFILE]:
   (and (exists MODULE  ?m    suchthat (ancestor [?m ?f]))
        (forall AFILE   ?a    suchthat (linkto [?m.afiles ?a])))
   :
   (?f.status = Compiled)        # back chain to compile

   { ARCHIVER update ?a.afile ?a.history ?f.object_code }

   (?f.status = Archived);



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


ranlib[?a:AFILE]:
   :
   { ARCHIVER randomize ?a.afile }
   ;


hide stuff [?a:AFILE]:
   (and (forall MODULE ?m suchthat (linkto [?m.afiles ?a]))
        (forall CFILE ?c suchthat (ancestor [?m ?c]))
        (forall YFILE ?y suchthat (ancestor [?m ?y]))
        (forall LFILE ?l suchthat (ancestor [?m ?l]))):
  { ARCHIVER stuff ?a.afile ?c.object_code ?y.object_code ?l.object_code}
  ;


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

```
strategy build

# This strategy defines the tool to build a PROGRAM, and it provides a rule
# to build a PROJECT as well.

imports data_model;
exports all;

objectbase

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

end_objectbase

rules

# Build the entire GROUP -- This is essentially a makep

build [?g:GROUP]:
   (forall PROJECT ?p suchthat (member [?g.projects ?p]))
   :
   (?p.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 BIN ?b suchthat (and (member [?proj.bin ?p])
#                                  (?b.arch = CurrentArch)))
    (forall BIN ?b suchthat  (member [?proj.bin ?b]))
    :
    (?b.build_status = Built)    # back chain to build BIN
    {}
    (?proj.build_status = Built);


# NOTE:
# it is assumed that the executable of the miniproject has links to
# the associated libraries.
#
```

```
build [?mp:MINIPROJECT]:
    (and (forall CFILE    ?c   suchthat (member [?mp.cfiles ?c]))
 (forall LFILE    ?x   suchthat (member [?mp.lfiles ?x]))
 (forall YFILE    ?y   suchthat (member [?mp.yfiles ?y]))
         (exists EXEFILE  ?e   suchthat (member [?mp.exec   ?e]))
         (forall AFILE    ?a   suchthat (linkto [?e.afiles  ?a])))
    :
    (and (?c.status = Compiled)
         (?x.status = Compiled)
         (?y.status = Compiled))

    { BUILD build_local ?c.object_code ?x.object_code ?y.object_code
?a.afile ?e.exec ?e.history }
    ;


# Used to backward chain to build on each executable.

build [?b:BIN]:
   (forall EXEFILE ?exe suchthat (member [?b.binexecs ?exe]))
    :
   (?exe.build_status = Built) # back chain to build EXEFILE
   {}
   (?b.build_status = Built);

# activation. if all libraries that this executable is linked to are
# marked as archived, build this executable using the libraries

build [?exe:EXEFILE] :
    (forall AFILE ?a suchthat (linkto [?exe.afiles ?a])):

    ( ?a.archive_status = Archived )      # back chain to archive AFILE

    { BUILD build_program ?exe.exec ?a.afile }

    ( ?exe.build_status = Built );
    ( ?exe.build_status = NotBuilt );


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

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

strategy cmarvel_chaining

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

imports data_model,
touch,
archive,
build,
compile,
        local,
print,
edit,
doc,
rcs,
mail,
marvel;

exports all;


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



```
# This file contains MSL commands to build an environment for developing
# C programs using a maximal amount of chaining amongst the rules.  In
# addition, all the rules available in cmarvel are contained here.  If
```

```
# a user only needed a subset of these rules, an master file similar to
# this one could be created that contains just the appropriate subset.

strategy cmarvel_chaining

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

imports data_model;
exports all;

rules


reserve[?f:FILE]:
    :
    { }

    no_forward (?f.reservation_status = CheckedOut);


# use forall, so these rules can be called from the local area.
edit[?c:CFILE]:
   (and (forall MODULE      ?m    suchthat (member   [?m.cfiles      ?c]))
        (forall MINIPROJECT ?mp   suchthat (member   [?mp.cfiles     ?c]))
(forall AFILE       ?a    suchthat (linkto   [?mp.link_afile ?a]))):

    # if the file has been reserved, you can go ahead and edit it

    (?c.reservation_status = CheckedOut)

    { }
    ;


arch[?l:LIB]:
 (forall AFILE ?a suchthat (member [?l.afiles ?a])):
 (?a.archive_status = Archived)
 { }
 (?l.archive_status = Archived);


arch[?m:MODULE]:
    (and (forall CFILE ?f suchthat (member [?m.cfiles ?f]))
 (forall YFILE ?y suchthat (member [?m.yfiles ?y]))
```

```
 (forall LFILE ?x suchthat (member [?m.lfiles ?x]))
 (forall MODULE ?child suchthat (member [?m.modules ?child])))
    :
    (and
         (?f.status = Archived)    # back chain to arch CFILE
         (?y.status = Archived)    # back chain to arch YFILE
         (?child.archive_status = Archived)) # back chain to arch MODULE
    {}
    (?m.archive_status = Archived);



#-----------------------------------------------------------------------
#
#
#                   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;
    lex_compile : string = lex_compile;
    yacc_compile : string = yacc_compile;
end


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

```
end_objectbase


rules

## this runs YACC on a file

compile [?y:YFILE]:
    (and (exists PROJECT ?p  suchthat (ancestor [?p ?y]))
         (forall INC     ?i  suchthat (member [?p.include ?i]))
         (forall HFILE   ?h  suchthat (member [?i.hfiles ?h])))
    :

    no_forward (?i.archive_status = Archived)

    { COMPILER yacc_compile ?y.contents ?y.object_code ?y.history ?y.ytabh
    ?h.contents "-g" }

    (and (?y.status = Compiled)
 (?y.object_time_stamp = CurrentTime));
    ( ?y.status = ErrorCompile );


# This runs Lex on the file

compile [?x:LFILE]:

    (and (exists PROJECT ?p  suchthat (ancestor [?p ?x]))
         (forall INC     ?i  suchthat (member   [?p.include ?i]))
         (forall HFILE   ?h  suchthat (member   [?i.hfiles  ?h]))
 (exists YFILE   ?y  suchthat (linkto   [?x.ytabh   ?y])))

    # 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 no_forward (?i.archive_status = Archived)
         (?y.status = Compiled))

    { COMPILER lex_compile ?x.contents ?x.object_code ?h.contents ?y.ytabh
   ?x.history "-g"}

    (and (?x.status = Compiled)
 (?x.object_time_stamp = CurrentTime));
```

```
    ( ?x.status = ErrorCompile);


compile [?f:CFILE]:

    (and (exists PROJECT ?p  suchthat (ancestor [?p ?f]))
         (forall INC     ?i  suchthat (member   [?p.include ?i]))
         (forall HFILE   ?h  suchthat (member   [?i.hfiles  ?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 no_forward (?i.archive_status = Archived)
         (?f.status = Analyzed))

     { COMPILER compile ?f.contents ?f.object_code ?h.contents
                        ?f.history "-g" }

    (and (?f.status = Compiled)
 (?f.object_time_stamp = CurrentTime));
    (?f.status = ErrorCompile);


analyze[?f:CFILE]:

    (and (exists PROJECT ?p  suchthat (ancestor [?p ?f]))
         (forall INC     ?i  suchthat (member   [?p.include ?i]))
         (forall HFILE   ?h  suchthat (member   [?i.hfiles  ?h]))):

    (and no_forward (?i.archive_status = Archived)
 (or no_backward (?f.status = New)
     no_backward (?f.status = NotAnalyzed)
     no_backward (?f.status = ErrorAnalyze)))

    { ANALYZER analyze ?f.contents ?h.contents ?f.history }

    (?f.status = Analyzed);
    (?f.status = ErrorAnalyze);


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

```
#
#                                  Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# doc startegy:  Contains rules for formatting document files
#

strategy doc

imports data_model;
exports all;

objectbase

  DOC_TOOLS :: superclass TOOL;
     Idraw   : string = Idraw;
     display_dvi  : string = display;
     format_latex : string = format_latex;
  end

end_objectbase

rules

display[?Doc:DOCFILE]:
   :
   (?Doc.reformat = false)

   { DOC_TOOLS display_dvi ?Doc}

   ;

format[?Doc:DOCFILE]:
   (forall POSTSCRIPT ?p suchthat (member [?Doc.postscript ?p])):
   (?Doc.reformat = true)

   { DOC_TOOLS format_latex ?Doc.contents ?Doc.output ?p.contents }

   (?Doc.reformat = false);


hide clean_post[?p:POSTSCRIPT]:
   (exists DOCFILE ?d suchthat (member [?d.postscript ?p])):
```

```
   (?d.reformat = false)
   { }
   (?p.reformat = false);


edit[?p:POSTSCRIPT]:
   :

   { DOC_TOOLS Idraw ?p.contents }

   (?p.reformat = true);
   (?p.reformat = false);



#-----------------------------------------------------------------------
#
#                   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, rcs;
exports all;

objectbase

EDITOR :: superclass TOOL;
    edit : string = editor;
    editnt : string = editor_no_tags;
end

VIEWER :: superclass TOOL;
    viewHist : string = viewHist;
    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.

# use forall, so these rules can be called from the local area.
edit[?c:CFILE]:
   (and (forall MODULE      ?m    suchthat (member   [?m.cfiles      ?c]))
        (forall MINIPROJECT ?mp   suchthat (member   [?mp.cfiles     ?c]))
(forall AFILE       ?a    suchthat (linkto   [?mp.link_afile ?a]))):

    # if the file has been reserved, you can go ahead and edit it

    (and (?c.reservation_status = CheckedOut)
 (?c.locker = CurrentUser))

    { EDITOR edit ?c.status ?c.contents ?c.history ?a.tags }

    (and (?c.status = NotAnalyzed)
 (?c.time_stamp = CurrentTime));
    (?c.reservation_status = CheckedOut);   # Dummy Postcondition

edit[?x:LFILE]:
   (and (exists MODULE ?m         suchthat (member   [?m.lfiles      ?x]))
        (forall MINIPROJECT ?mp   suchthat (member   [?mp.lfiles     ?x]))
(forall AFILE       ?a    suchthat (linkto   [?mp.link_afile ?a]))):

    # if the file has been reserved, you can go ahead and edit it

    (and (?x.reservation_status = CheckedOut)
 (?x.locker = CurrentUser))

    { EDITOR edit ?x.status ?x.contents ?x.history ?a.tags }

    (and (?x.status = NotCompiled)
   (?x.time_stamp = CurrentTime));
```

```
edit[?y:YFILE]:
   (and (exists MODULE      ?m    suchthat (member   [?m.yfiles      ?y]))
        (forall MINIPROJECT ?mp   suchthat (member   [?mp.yfiles     ?y]))
(forall AFILE       ?a    suchthat (linkto   [?mp.link_afile ?a]))):

    # if the file has been reserved, you can go ahead and edit it

    (and (?y.reservation_status = CheckedOut)
 (?y.locker = CurrentUser))

    { EDITOR edit ?y.status ?y.contents ?y.history ?a.tags }

    (and (?y.status = NotCompiled)
 (?y.time_stamp = CurrentTime));


# this edit rule is for editing include files.

edit[?h:HFILE]:

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

    { EDITOR editnt ?h.contents ?h.history }

    (and (?h.recompile_mod = true)
 (?h.time_stamp = CurrentTime));

# This edit rule is for editing LaTeX documents

edit[?d:DOCFILE]:

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

    { EDITOR editnt ?d.contents ?d.history }

    (and (?d.reformat = true)
 (?d.time_stamp = CurrentTime));


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

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


viewHist[?h:HISTORY]:
    :
    { VIEWER viewHist ?h.history }
    ;


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


strategy local

# 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


LOCAL :: superclass TOOL;
    compile      : string = compile;
    lex_compile  : string = lex_compile;
    yacc_compile : string = yacc_compile;
    analyze      : string = analyze;
end

end_objectbase
```

```
rules

lcompile [?y:YFILE]:

    (and (exists MINIPROJECT ?mp  suchthat (member [?mp.yfiles     ?y]))
         (forall AFILE       ?a   suchthat (linkto [?mp.link_afile ?a]))
 (forall PROJECT     ?p   suchthat (ancestor [?p ?a]))
         (forall INC         ?i   suchthat (or (member [?p.include  ?i])
                                               (linkto [?p.link_inc ?i])))
 (forall HFILE       ?h   suchthat (or (member  [?mp.hfiles ?h])
                                                (member [?i.hfiles  ?h])))):

    no_forward (?i.archive_status = Archived)

    { LOCAL yacc_compile ?y.contents ?y.object_code ?h.contents ?y.history "-g" }

    (and (?y.status = Compiled)
 (?y.object_time_stamp = CurrentTime));
    (?y.status = ErrorCompile);


# This runs Lex on the file

lcompile [?x:LFILE]:

    (and (exists MINIPROJECT ?mp  suchthat (member [?mp.lfiles     ?x]))
         (forall AFILE       ?a   suchthat (linkto [?mp.link_afile ?a]))
 (forall PROJECT     ?p   suchthat (ancestor [?p ?a]))
         (forall INC         ?i   suchthat (or (member [?p.include  ?i])
                                               (linkto [?p.link_inc ?i])))
 (forall HFILE       ?h   suchthat (or (member [?mp.hfiles ?h])
                                               (member [?i.hfiles  ?h])))
         (exists YFILE       ?y   suchthat (linkto [?x.ytabh ?y]))):

    (and no_forward (?i.archive_status = Archived)
 (?y.status = Compiled))

    { LOCAL lex_compile ?x.contents ?x.object_code ?h.contents
    ?y.ytabh ?x.history "-g"}

    (and (?x.status = Compiled)
 (?x.object_time_stamp = CurrentTime));
    ( ?x.status = ErrorCompile);
```

```
lcompile [?f:CFILE]:

    (and (exists MINIPROJECT ?mp  suchthat (member [?mp.cfiles     ?f]))
         (forall AFILE       ?a   suchthat (linkto [?mp.link_afile ?a]))
 (forall PROJECT     ?p   suchthat (ancestor [?p ?a]))
         (forall INC         ?i   suchthat (or (member [?p.include ?i])
                                               (linkto [?p.link_inc ?i])))
 (forall HFILE       ?h   suchthat (or (member [?mp.hfiles ?h])
                                               (member [?i.hfiles  ?h])))):
    (and no_forward (?i.archive_status = Archived)
         (?f.status = Analyzed))


     { LOCAL compile ?f.contents ?f.object_code ?h.contents ?f.history"-g" }

    (and (?f.status = Compiled)
 (?f.object_time_stamp = CurrentTime));
    (?f.status = ErrorCompile);


lanalyze[?f:CFILE]:

    (and (exists MINIPROJECT ?mp  suchthat (member [?mp.cfiles     ?f]))
         (forall AFILE       ?a   suchthat (linkto [?mp.link_afile ?a]))
 (forall PROJECT     ?p   suchthat (ancestor [?p ?a]))
         (forall INC         ?i   suchthat (or (member [?p.include  ?i])
                                               (linkto [?p.link_inc ?i])))
 (forall HFILE       ?h   suchthat (or (member [?mp.hfiles ?h])
                                               (member [?i.hfiles  ?h])))):

    (and no_forward (?i.archive_status = Archived)
         (or no_backward (?f.status = New)
     no_backward (?f.status = NotAnalyzed)
     no_backward (?f.status = ErrorAnalyze)))

    { LOCAL analyze ?f.contents ?h.contents ?f.history }

    (?f.status = Analyzed);
    (?f.status = ErrorAnalyze);


#----------------------------------------------------------------------
#
```

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

strategy mail

# This strategy contains rules to send mail to the Marvel Developers
#
imports data_model;
exports all;

objectbase

MAIL :: superclass TOOL;
    mail : string = mail_marvel;
end

end_objectbase

rules

mail[?P:PROGRAMMER]:
  :
  { MAIL mail ?P.mail_id }
  ;



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

# This strategy contains rules for starting and halting the Marvel Programs
#
```

```
strategy marvel

imports data_model;
exports all;

objectbase

MARVEL  :: superclass TOOL;
   marvel_go : string = marvel_go;
   marvel_local_go : string = marvel_local_go;
end

end_objectbase

rules

exec_local[?p:PROGRAMMER]:
   (and (exists EXEFILE ?s suchthat (linkto [ ?p.server ?s ]))
        (exists EXEFILE ?c suchthat (linkto [ ?p.client ?c ]))
        (exists EXEFILE ?l suchthat (linkto [ ?p.loader ?l ])))
   :
   { MARVEL marvel_local_go ?s.exec ?c.exec ?l.exec }
   ;


exec_marvel[?s:EXEFILE, ?c:EXEFILE, ?l:EXEFILE]:
  :
  { MARVEL marvel_go ?s.exec ?c.exec ?l.exec }
  ;

#-----------------------------------------------------------------------
#
#                    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     : string = print;
    print_dvi : string = print_dvi;
end

end_objectbase

rules

printer [?f:FILE]:
  :
  { PRINT print ?f.contents }
  ;

printer [?d:DOCFILE]:
  (forall POSTSCRIPT ?p suchthat (member [?d.postscript ?p]))
  :
  (?d.reformat = false)
  { PRINT print_dvi ?d.output ?p.contents }
  ;

printer [?p:POSTSCRIPT]:
  :
  { PRINT print ?p.contents }
  ;

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

# 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;
    deposit : string = check_in;
    replace : string = replace;
    create  : string = check_in_create;
    locked  : string = locked;
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]:
    :
    no_forward (?f.reservation_status = Available)    # backward chain to create_rcs

    { RCS reserve ?f.contents ?f.version ?f.history }

    (and no_forward (?f.reservation_status = CheckedOut)
            (?f.time_stamp        = CurrentTime)
                    (?f.locker            = CurrentUser));

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

deposit[?old:FILE]:
    :
    (and [?old.locker = CurrentUser]
         [?old.reservation_status = CheckedOut])

    { RCS deposit ?old.contents ?old.version ?old.history }

    [?old.reservation_status = Available];
```

```
deposit[?old:HFILE]:
    :
    (and [?old.locker = CurrentUser]
         [?old.reservation_status = CheckedOut])

    { RCS deposit ?old.contents ?old.version ?old.history }

    (and [?old.reservation_status = Available]
 (?old.recompile_mod = false));



create_rcs[?f:FILE]:
    :
    [?f.reservation_status = None]

    { RCS create ?f.contents ?f.version }

    (?f.reservation_status = Available);



replace[?new:FILE, ?old:FILE]:
    :
    (and [?old.locker = CurrentUser]
         [?old.reservation_status = CheckedOut])

    { RCS replace ?new.contents ?old.version ?new.history ?old.history }

    [?old.reservation_status = Available];


locked[?s:SRC]:
   (forall CFILE ?c suchthat (and (ancestor [?s ?c])
  (?c.reservation_status = CheckedOut))):
   { RCS locked ?c.contents ?c.locker }
   ;

#----------------------------------------------------------------------

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

```
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#
#
# This strategy contains various "touch" rules used
# in consistency (forward) chaining to propagate modifications to
# other objects.
# touchup set of rules propagate changes up the
# hierarchy whenever a source file is modified.
# touchdw set of rules propagate changes down the hierarchy due to
# a modification in a header file.
# NOTE: Once we have the capability to
# make assertions on bound variables, we won't need the touchdw
# rules.
#

strategy clean

imports data_model;
exports all;

rules

hide touchup[?Proj:PROJECT]:
    (exists SRC ?s suchthat (member [?Proj.src ?s]))
    :
    [?s.archive_status = NotArchived]  # from touch SRC
    {}
    [?Proj.build_status = NotBuilt];


hide touchup[?s:SRC]:
    (exists MODULE ?m suchthat (member [?s.modules ?m]))
    :
    [?m.archive_status = NotArchived]  # from touch MODULE
    {}
    [?s.archive_status = NotArchived];


hide touchup[?b:BIN]:
    (exists PROJECT ?Proj suchthat (member [?Proj.bin ?b])):
    [?Proj.build_status = NotBuilt]      # from touch PROJECT
    {}
    [?b.build_status = NotBuilt];
```

```
hide touchup[?e:EXEFILE]:
    (exists BIN ?b suchthat (member [?b.binexecs ?e])):
    [?b.build_status = NotBuilt]      # from touch BIN
    {}
    [?e.build_status = NotBuilt];

# This rule will "touch" a library if any of its modules have become
# "NotArchived."
# we call it touchupL due to a bug in adding forward pointers
# in compile_chain_network.

hide touchupL[?l:LIB]:
   (exists PROJECT ?p suchthat (member [?p.lib ?l]))
   :
   [?p.build_status = NotBuilt]           # from touch PROJECT
   {}
   [?l.archive_status = NotArchived];


# This rule will "touch" a module if any of its CFILEs has become
# "NotCompiled" or compiled in "Error".
# also, if the reservation status became available
# as a result of deposit touch the module.

hide touchup[?M:MODULE]:
   (and
    (forall CFILE  ?c suchthat (member [?M.cfiles ?c]))
    (forall YFILE  ?y suchthat (member [?M.yfiles ?y]))
    (forall LFILE  ?x suchthat (member [?M.lfiles ?x]))
    (forall MODULE ?m suchthat (member [?M.modules ?m])))
    :
    (or [?c.status     = ErrorCompile]         # from compile CFILE
        [?c.reservation_status = Available]     # from deposit FILE
        [?y.status     = ErrorCompile]         # from compile YFILE
        [?x.status     = ErrorCompile]         # from compile LFILE
        [?m.archive_status     = NotArchived])
    {}
    [?m.archive_status = NotArchived];


# below is the set of touch down rules, with the exception of touch on
# inc file which actually initiates them.
#
```

```
hide touchup[?i:INC]:
   (exists HFILE ?h suchthat (member [?i.hfiles ?h]))
   :
   [?h.recompile_mod = true]    # from edit hfile
   {}
   [?i.archive_status = INC_NotArchived];

hide touchdw[?p:PROJECT]:
   (exists INC ?i  suchthat (member [?p.include ?i]))
   :
   [?i.archive_status = INC_NotArchived]
   {}
   [?p.build_status = INC_NotBuilt];

hide touchdw[?s:SRC]:
   (exists PROJECT ?p  suchthat (member [?p.src ?s]))
   :
   [?p.build_status = INC_NotBuilt]
   {}
   [?s.archive_status = INC_NotArchived];

hide touchdw[?m:MODULE]:
   (exists SRC ?s  suchthat (member [?s.modules ?m]))
   :
   [?s.archive_status = INC_NotArchived]
   {}
   [?m.archive_status = INC_NotArchived];


hide touchdw[?c:CFILE]:
   (exists MODULE ?m  suchthat (member [?m.cfiles ?c]))
   :
   [?m.archive_status = INC_NotArchived]
   {}
   [?c.status = NotCompiled];
```

B.3 C/Marvel Envelopes

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

```
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# Idraw envelope
#
# usage: Idraw [POSTSCRIPT]
#


ENVELOPE Idraw;

SHELL sh;

INPUT
text : thefig;
OUTPUT
none ;

BEGIN

set -x

FILENAME=`basename $thefig`

echo
echo Executing Idraw on figure in $FILENAME ...

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

if [ -f $thefig ]
then
   touch $thefig
fi

idraw $thefig

# Check to make sure that the file really existed.
#
```

```
if [ $Created = "YES" ]
then
   echo "Figure $FILENAME Created."
   RETURN_CODE=0
else
   NewReport=`ls -l $thefig`

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

RETURN "$RETURN_CODE";
END

#----------------------------------------------------------------------
#
#                    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] [HFILE...] [CFILE.history]
#

ENVELOPE analyze;

SHELL ksh;

INPUT
text        : thefile;
set_of INC  : ifiles;
text     : history;
OUTPUT
none ;
```

```
BEGIN

header=`get_dirname $thefile`
shortname=`basename $thefile`

log=$history

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

tmp_dir=/tmp/analyze$$
mkdir $tmp_dir

# we need to make the -I list
#
idir=""
for i in $ifiles
do
   ln -s $i $tmp_dir
done
idir="-I$tmp_dir"

echo "lint $CCFLAGS -c $idir $shortname -ll -lc -lm -lX11"
echo "lint $CCFLAGS -c $idir $shortname -ll -lc -lm -lX11" >> $log

# place the output in temporary place, since
# we will clean the output up.
# -----------------------------------------
lint_output=/tmp/analyze.2.$$

lint $idir $thefile >> $lint_output 2>&1
lint_status=$?

# Reduces the output by removing unnecessary header information
#
cat $lint_output | sed s?$header/?? >> $log
rm $lint_output

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

```
if [ $lint_status -eq 0 ]
then
    echo lint successful, results available with viewHist on $log
    echo lint successful >> $log
    RET_VAL=0
else
    echo lint failed, results available with viewHist on $log
    echo lint failed >> $log
    RET_VAL=1
fi

RETURN "$RET_VAL" ;
END

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

ENVELOPE build;

SHELL sh;

INPUT
binary    : executable;
set_of LIB : libraries;
OUTPUT
none ;

BEGIN


shortname=`basename $executable`

echo "building $shortname on `date`"

if [ "x$libraries" = "x" ]
```

```
then
   echo "No libraries. Nothing to build"
   RET_VAL=1
else
   if [ -f $executable ]
   then
      rm $executable
   fi

   echo "cc {LIBRARIES} -L/usr/local/gnu/lib -ll -lc -lm -lX11 -lgdbm -o {EXECUTABLE}"
   cc $libraries -L/usr/local/gnu/lib -ll -lc -lm -lX11 -lgdbm -o $executable

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

   if [ ! -f $executable ]
   then
      echo "Build failed"
      RET_VAL=1
   else
      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_VAL=0
          else
      echo build failed      # Otherwise, notify the user that
      RET_VAL=1                  # that there must have been a load error
          fi
      else
          echo build failed
          RET_VAL=1
      fi
   fi
fi

RETURN "$RET_VAL";
END

#-----------------------------------------------------------------------
#
#                   Marvel Software Development Environment
#
#                                Copyright 1991
#                     The Trustees of Columbia University
#                            in the City of New York
#                              All Rights Reserved
#
# build_local envelope
#
# usage: build_local [CFILE.object_code] [LFILE.object_code]
#      [YFILE.object_code] [AFILE.afile] [EXEFILE.exec]

ENVELOPE build_local;

SHELL sh;

INPUT
set_of binary : c_objects;
set_of binary : x_objects;
set_of binary : y_objects;
set_of binary : libraries;
binary : executable;
OUTPUT
none ;

BEGIN
```

```
set -x

shortname=`basename $executable`

echo "locally building $shortname on `date`"

local_files="$c_objects $x_objects $y_objects"

echo "cc {LOCAL_FILES} {LIBRARIES} -ll -lc -lm -lX11 -lgdbm -o $shortname"

cc $local_files $libraries -L/usr/local/gnu/lib -ll -lc -lm -lX11 -lgdbm -o $executable

# 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_VAL=0
    else
```

```
        echo loader failed    # Otherwise, notify the user that
        chmod o-x $executable # that there must have been a load error
        RET_VAL=1             # since the executable bit was not set
    fi
else
    echo build failed
    RET_VAL=1
fi

RETURN "$RET_VAL";
END

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

ENVELOPE check_in;

SHELL ksh;

INPUT
text : object;
text : rcs_file;
text : history;
OUTPUT
none ;

BEGIN

name=`basename $object`

echo "depositing $name on `date`"
echo "depositing $name on `date`" >> $history

# This script is used to check in (deposit) code.
# It is a simple end to the RCS ci command.
```

```
echo "check in $object  [y/n]?"
read ans

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

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

   if [ "x$message" = "x" ]
   then
      ci -u $rcs_file $object
   else
      ci -m"$message" -u $rcs_file $object
   fi

#   rm -f $object
   co $rcs_file $object

   echo $name deposited
   echo $name deposited >> $history
   RET_VAL=0
else
   echo $name NOT deposited
   echo $name NOT deposited >> $history
   RET_VAL=1
fi

RETURN "$RET_VAL";
END

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

```
SHELL sh;

INPUT
text : object;
text : rcs_file;
OUTPUT
none ;
BEGIN


# This script is used to check in (deposit) code for the first time
# It is a simple front end for the e RCS ci command.

name=`basename $object`
echo Now entering $name into the Revision Control System.

  # Since 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 the
  # file is there.

touch $object
touch $rcs_file

ci -l $rcs_file $object

echo an RCS file for $name has been created.

RETURN "0";
END

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

ENVELOPE check_out;
```

```
SHELL ksh;

INPUT
text : object;
text : rcs_file;
text : history;
OUTPUT
none ;
BEGIN

name=`basename $object`

echo "reserving $name on `date`"
echo "reserving $name on `date`" >> $history

if [ ! -f $rcs_file ]
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 $object ]
   then
      echo New File, nothing to reserve.  Creating RCS file
      echo New File, nothing to reserve.  Creating RCS file >> $history
      touch $object
      touch $rcs_file
      RET_VAL=0
   else
      ci -l $rcs_file $object
      echo $name reserved
      echo $name reserved >> $history
   fi
   RET_VAL=0
else
   echo "check out $name [y/n]? "
   read ans

   if [ "x$ans" = "xy" ]
   then
      co -l $rcs_file $object
      echo $name reserved
      echo $name reserved >> $history
      RET_VAL=0
   else
```

```
      echo Reservation cancelled at user's request.
      echo Reservation cancelled at user's request. >> $history
      echo $name not reserved
      echo $name not reserved >> $history
      RET_VAL=1
   fi
fi

RETURN "$RET_VAL";
END

#----------------------------------------------------------------------
#
#                    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]
#
ENVELOPE compile;

SHELL sh;

INPUT
text       : thefile;
binary     : obj_file;
set_of INC : ifiles;
text       : history;
literal    : CCFLAGS;
OUTPUT
none ;
BEGIN

shortname=`basename $thefile`
shortobj=`basename $obj_file`

log=$history

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

```
echo
echo >> $log

tmp_dir=/tmp/compile$$
mkdir $tmp_dir

# we need to make the -I list
#
idir=""
for i in $ifiles
do
   ln -s $i $tmp_dir
done
idir="-I$tmp_dir"

echo "cc $CCFLAGS -c $idir $shortname -o $shortobj -ll -lc -lm -lX11"
echo "cc $CCFLAGS -c $idir $shortname -o $shortobj -ll -lc -lm -lX11" >> $log
cc $CCFLAGS -c $idir $thefile -o $obj_file -ll -lc -lm -lX11 >> $log 2>&1
cc_status=$?

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

if [ $cc_status -eq 0 ]
then
    echo compile successful, results available with viewHist
    echo compile successful >> $log
    RET_VAL=0
else
    echo compile failed, results available with viewHist
    echo compile failed >> $log
    RET_VAL=1
fi

RETURN "$RET_VAL";
END

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

ENVELOPE display_dvi;

SHELL ksh;

INPUT
        binary       : thedvi;
OUTPUT
none ;
BEGIN

xdvi $thedvi &

RETURN "0" ;
END

#-----------------------------------------------------------------------
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# editor envelope
#
# usage: edit [FILE.status] [FILE] [FILE.history] [TAGS...]
#
# This edits the chosen file, and sends along the library which has
# power over it so that emacs will read in the TAGS file associated
# with it.  This also incorporates a simple locking mechanism by
```

```
# making the file writable when it edits it, and removing this capability
# when it leaves.

ENVELOPE editor;

INPUT
    enumerated : status;
          text : thefile;
          text : history;
   set_of text : link_tags_file;

OUTPUT
none ;
BEGIN

shortname=`basename $thefile`

echo Editing source file in $basename on `date`
echo Editing source file in $basename on `date` >> $history
echo
echo >> $history


# 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
```

```
     tags="$link_tags_file"
     for i in $tags
     do
       echo "(visit-tags-table \"$i\")" >> $EL_FILE
     done

     echo "(switch-to-buffer \"$shortname\")" >> $EL_FILE

     if [ $status != "Analyzed" ] && [ $status != "Compiled" ] && [ $status != Archived ]
     then
        echo \(find-file-read-only \"$history\"\) >> $EL_FILE
        echo "(split-window)" >> $EL_FILE
        echo "(switch-to-buffer \"$shortname\")" >> $EL_FILE
     fi

     $EDITOR $thefile -l $EL_FILE

     rm $EL_FILE
else
     vi $thefile
fi


RET_VAL=0

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

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

```
fi

RETURN "$RET_VAL" ;
END

#----------------------------------------------------------------------
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# editor_no_tags envelope
#
# usage: edit [FILE]
#


ENVELOPE

INPUT
text : thefile;
text : history;
OUTPUT
none ;

BEGIN

shortname=`basename $thefile`

echo Editing source file in $basename on `date`
echo Editing source file in $basename on `date` >> $history
echo
echo >> $history


# 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
     $EDITOR $thefile
else
     vi $thefile
fi


RET_VAL=0

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

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

RETURN "$RET_VAL" ;
END

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

```
#
#                                Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# runs latex on the given document.
#
ENVELOPE format_latex;

SHELL ksh;

INPUT
text        : the_latex_file;
        binary      : the_dvi_file;
set_of text : figures;
OUTPUT
none ;

BEGIN

FILENAME=`basename $the_latex_file`
DIRNAME=`get_dirname $the_latex_file`
current=$PWD

# Let latex know where the included figures are going to be
# ----------------------------------------------------------
tmp_dir=/tmp/format_latex$$
mkdir $tmp_dir

# link to all the included figures, and set
# the appropriate TEXINPUTS shell variable to
# let dvips know where these encapsulated figures
# are kept.

idir=""
for i in $figures
do
   ln -s $i $tmp_dir
done
idir="-I$tmp_dir"
TEXINPUTS="$TEXINPUTS:$tmp_dir"

# Silly necessity to check if there is already
# an extension of .tex
```

```
# -------------------------------------------
if [ "x`getsuffix $the_latex_file`" = "xtex" ]
  latex $the_latex_file
  FILENAME=`getname $FILENAME`
else
  latex $the_latex_file.tex
fi

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

mv $current/$FILENAME.dvi $the_dvi_file

echo "Cleaning up temp files"
rm $FILENAME.aux
rm $FILENAME.log
rm $FILENAME.bbl
rm $FILENAME.blg

RETURN "0";
END

#----------------------------------------------------------------------
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# lex_compile envelope
#
# usage: lex_compile [LFILE.contents] [LFILE.object_code] [HFILE.contents ...]
#                    [YTABH.contents] [LFILE.history] {LITERAL}

ENVELOPE lex_compile;

SHELL sh;

INPUT
          text      : thefile;
          binary    : obj_file;
```

```
   set_of INC       : ifiles;
          text      : ytabh;
          text      : history;
       literal      : CCFLAGS;
OUTPUT
none ;

BEGIN

shortname=`basename $thefile`
short_obj=`basename $obj_file`
log=$history

echo "running lex on $shortname on `date`"
echo "running lex on $shortname on `date`" >> $log
echo
echo >> $log


tmp_dir=/tmp/compile$$
mkdir $tmp_dir

# we need to make the -I list
# ----------------------------------
idir=""
if [ "x$ifiles" != "x" ]
then
    ln -s $ifiles $tmp_dir
    idir="-I$tmp_dir"
fi

# insert the y.tab.h header file also
# ----------------------------------
ln -s $ytabh $tmp_dir/y.tab.h

echo "lex -t $shortname > lexer.c"
echo "lex -t $shortname > lexer.c" >> $log
lex -t $thefile > lexer.c

echo "cc $CCFLAGS -c $idir $shortname -o $short_obj -ll -lc -lm -lX11"
echo "cc $CCFLAGS -c $idir $shortname -o $short_obj -ll -lc -lm -lX11" >> $log
cc $CCFLAGS -c $idir lexer.c -o $obj_file -ll -lc -lm -lX11 >> $log 2>&1
cc_status=$?

echo "rm -f lexer.c" >> $log
```

```
rm -f lexer.c

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

if [ $cc_status -eq 0 ]
then
    echo compile successful, results available with viewHist
    echo compile successful >> $log
    RET_VAL=0
else
    echo compile failed, results available with viewHist
    echo compile failed >> $log
    RET_VAL=1
fi

RETURN "$RET_VAL";
END

#----------------------------------------------------------------------
#
#                    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 [AFILE]

ENVELOPE list_archive;

INPUT
binary : afile;
OUTPUT
none ;
BEGIN

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

```
RETURN "0";
END

#-----------------------------------------------------------------------
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# locked envelope
#
# usage: locked [SRC]
#

ENVELOPE locked;

SHELL ksh;

INPUT
set_of text   : thefiles;
set_of string : the_lockers;
OUTPUT
none ;
BEGIN

set -x

echo "The following files are currently locked:"

let num=0
LOCKED_LIST=
for f in $thefiles
do
  set -A LOCKED_LIST num $f
  let num=$num+1
done
let total=$num

let num=$num+1
for i in $the_lockers;
do
  set -A LOCKED_LIST num $i
```

```
done

RET_VAL=0
let ct=0
for i in $library
do
    LINE="`basename ${LOCKED_LIST[$ct]}` locked by"
    let nt=$ct+$total
    LINE="$LINE ${LOCKED_LIST[$nt]}"

    echo $LINE
done

RETURN "0";
END

#----------------------------------------------------------------------
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# mail envelope
#
# usage: mail [PROGRAMMER.mail_id]
#

ENVELOPE mail;

SHELL ksh;

INPUT
string : id;
OUTPUT
none ;
BEGIN

echo "Enter in your letter, press <ctrl>-d to send."

mail $id@cs.columbia.edu
if [ $? -eq 0 ]
then
```

```
   echo "Mail sent..."
else
   echo "Mail cancelled."
fi

RETURN "0";
END

#----------------------------------------------------------------------
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# marvel_go envelope
#
# usage: marvel_go [GROUP...]
#
# function: This will start the Server and Client and remove them when
#           the user is finished.
#
ENVELOPE

SHELL sh;

INPUT
binary : Server;
binary : Client;
binary : Loader;
OUTPUT
none ;

BEGIN

echo "What directory are you using: "
read direct

cd $direct

if [ $? -ne 0 ] || [ "$directx" = "x" ]
then
  echo "BAD DIRECTORY!"
```

```
else
   #----------------------------------------------------------
   # sh constructs
   #
    export MARVEL_ROOT
    export MARVEL_HELP_DIR
    export MARVEL_LOADER

    MARVEL_ROOT=`pwd`
    MARVEL_HELP_DIR=`pwd`
    MARVEL_LOADER=`echo $Loader`

   #----------------------------------------------------------

   if [ -f .server_port ]
   then
      where=`tail -1 .server_port`

      echo "There already is a server running."
      echo "Connecting to $where"
      thispid=
      where="-h $where"
   else
      # Start the server, and remember process ID
      # so it can be killed later
      # ----------------------------------------
      xterm -rv -T SERVER -n server -e $Server &
      thispid=$!
      where=

      echo "Please wait for the server program to run before continuing."
      echo "hit <ENTER> when server is running"
      read ans
   fi

   xterm -rv -T CLIENT -n client -e $Client -w $where

   # Kill the server
   # ---------------
   if [ "x$thepid" -ne "x" ]
   then
      kill $thispid
   fi
fi
```

```
RETURN "0";
END

#-----------------------------------------------------------------------
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# marvel_local_go envelope
#
# usage: marvel_local_go [PROGRAMMER]
#
# function: This will start the Server and Client and remove them when
#           the user is finished.
#
ENVELOPE

SHELL sh;

INPUT
set_of binary : Server;
set_of binary : Client;
set_of binary : Loader;
OUTPUT
none ;

BEGIN

echo "What directory are you using: "
read direct

cd $direct

if [ $? -ne 0 ] || [ "$directx" = "x" ]
then
  echo "BAD DIRECTORY!"
else
   #-----------------------------------------------------------
   # sh constructs
   #
    export MARVEL_ROOT
```

```
    export MARVEL_HELP_DIR
    export MARVEL_LOADER

    MARVEL_ROOT=`pwd`
    MARVEL_HELP_DIR=`pwd`
    MARVEL_LOADER=`echo $Loader`

   #-----------------------------------------------------------

   if [ -f .server_port ]
   then
      where=`tail -1 .server_port`

      echo "There already is a server running."
      echo "Connecting to $where"
      thispid=
      where="-h $where"
   else
      # Start the server, and remember process ID
      # so it can be killed later
      # ----------------------------------------
      xterm -rv -T SERVER -n server -e $Server &
      thispid=$!
      where=

      echo "Please wait for the server program to run before continuing."
      echo "hit <ENTER> when server is running"
      read ans
   fi

   xterm -rv -T CLIENT -n client -e $Client -w $where

   # Kill the server
   # ---------------
   if [ "x$thepid" -ne "x" ]
   then
      kill $thispid
   fi
fi

RETURN "0";
END

#-----------------------------------------------------------------------
#
```

```
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# Print Envelope: This prints the specified file to the default printer.
#

ENVELOPE print;

SHELL ksh;

INPUT
text : thefile;
OUTPUT
none ;
BEGIN

FILE=`basename $thefile`

echo "Now Printing $FILE..."
echo "Print to which Printer (or Press Return to Cancel)"

read ans?
if [ "x$ans" = "x" ]
then
   echo "Printing cancelled at user's request"
else
   echo "lpr -P$ans $FILE"
   lpr -P$ans $thefile
   if [ $? -ne 0 ]
   then
      echo "Errors encountered while printing $FILE"
   fi
fi

RETURN "0";
END

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

```
#                                 Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# Print_dvi Envelope: This prints the specified file to the default printer.
#

ENVELOPE print_dvi;

SHELL ksh;

INPUT
text     : thefile;
        set_of text : figures;
OUTPUT
none ;
BEGIN

FILE=`basename $thefile`

echo "Now Printing $FILE..."
echo "Print to which Printer (or Press Return to Cancel)"


read ans?
if [ "x$ans" = "x" ]
then
   echo "Printing cancelled at user's request"
else
   echo "lpr -P$ans $FILE"
   PRINTER=$ans

   tmp_dir=/tmp/print_dvi$$
   mkdir $tmp_dir

   # link to all the included figures, and set
   # the appropriate TEXINPUTS shell variable to
   # let dvips know where these encapsulated figures
   # are kept.

   idir=""
   for i in $figures
   do
      ln -s $i $tmp_dir
```

```
   done
   idir="-I$tmp_dir"
   TEXINPUTS="$TEXINPUTS:$tmp_dir"

   dvips $thefile
   if [ $? -ne 0 ]
   then
      echo "Errors encountered while printing $FILE"
   fi

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

RETURN "0";
END

#----------------------------------------------------------------------
#
#                    Marvel Software Development Environment
#
#                                 Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# randomize envelope
#
# usage: randomize [AFILE]
#
# function: Randomizes the library contained in AFILE.
#

ENVELOPE randomize;

INPUT
binary : afile;
text   : history;
OUTPUT
none ;
BEGIN

shortname=`basename $afile`
```

```
log=$history
MT=`arch`;

echo "randomizing $shortname on `date`"
echo "randomizing $shortname on `date`" >> $log
echo
echo >> $log

echo "ranlib $afile"
echo "ranlib $afile" >> $log
ranlib $afile >> $log
RET_VAL=$?

RETURN "$RET_VAL";
END

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

ENVELOPE replace;

SHELL ksh;

INPUT
text : object;        # new object
text : rcs_file;      # old versionable object
text : new_history;   # history of new object
        text : old_history;   # history of old object
OUTPUT
none ;

BEGIN

name=`basename $object`

echo "depositing $name on `date`"
```

```
echo "depositing $name on `date`" >> $new_history

echo "replacing `basename $rcs_file` with $name on `date`"
echo "replacing `basename $rcs_file` with $name on `date`" >> $old_history

# This script is used to check in (deposit) code.
# It is a simple end to the RCS ci command.

echo "check in $object  [y/n]?"
read ans

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

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

   if [ "x$message" = "x" ]
   then
      ci -u $rcs_file $object
   else
      ci -m"$message" -u $rcs_file $object
   fi

#    rm -f $object
   co $rcs_file $object

   echo $name deposited
   echo $name deposited >> $new_history
   echo $name deposited >> $old_history
   RET_VAL=0
else
   echo $name NOT deposited
   echo $name NOT deposited >> $history
   echo $name NOT deposited >> $old_history
   RET_VAL=1
fi

RETURN "$RET_VAL";
END

#----------------------------------------------------------------------
#
```

```
#                    Marvel Software Development Environment
#
#                                Copyright 1991
#                      The Trustees of Columbia University
#                             in the City of New York
#                               All Rights Reserved
#
# stuff envelope
#
# usage: stuff [AFILE.afile...] [AFILE.history...] [FILE.object_code]
#
# function: This takes all the object codes which belong to a certain
#           afile and replaces the old library with these objectcodes.
#

ENVELOPE update;

SHELL sh;

INPUT
binary        : library;
        set_of binary : cfiles;
        set_of binary : yfiles;
        set_of binary : lfiles;
OUTPUT
none ;

BEGIN

set -x
object_code="$cfiles $yfiles $lfiles"
ar rv $library $object_code

RETURN "0";
END

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

```
# update envelope
#
# usage: update [AFILE.afile...] [AFILE.history...] [FILE.object_code]
#
# function: This takes all the object codes which have been updated
#           and replaces their entries in the library.
#

ENVELOPE update;

SHELL sh;

INPUT
set_of binary  : library;
        set_of text    : history;
binary         : object_code;
OUTPUT
none ;

BEGIN

set -x

short_obj=`basename $object_code`
header_obj=`get_dirname $object_code`

RET_VAL=0

ct="x"
for the_lib in $library
do
    his="x"
    for the_hist in $history
    do
       if [ $ct -ne $his ]
       then
          his="$his"x
       else
          shortname=`basename $the_lib`
          header=`get_dirname $the_lib`
          MT=`arch`

          log=$the_hist
          if [ ! -f $log ]
          then
```

```
            touch $log
        fi

        echo "updating $shortname on `date`"
        echo "updating $shortname on `date`" >> $log
        echo
        echo >> $log

        echo "ar rv $shortname $short_obj"
        echo "ar rv $shortname $short_obj" >> $log

        TEMP_FILE=/tmp/update$$
        TEMP_FILE_2=/tmp/update_2$$

        ar rv $the_lib $object_code >> $TEMP_FILE
        update_status=$?

        cat $TEMP_FILE | sed s?$header/?? >> $TEMP_FILE_2
        cat $TEMP_FILE_2 | sed s?$header_obj/?? >> $log

        if [ -f $TEMP_FILE ]
        then
            rm $TEMP_FILE
        fi
        if [ -f $TEMP_FILE_2 ]
        then
            rm $TEMP_FILE_2
        fi

        if [ $update_status -eq 0 ]
        then
            echo "$short_obj is now archived in $library"
            echo "$short_obj is now archived in $library" >> $log
            echo
            echo >> $log
        else
            echo archive failed
            echo archive failed >> $log
            echo
            echo >> $log
            RET_VAL=1
        fi
    fi
done
ct="$ct"x
```

```
done

RETURN "$RET_VAL";
END

#-----------------------------------------------------------------------
#
#                     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]


ENVELOPE view;

SHELL ksh;

INPUT
text : thefile;
OUTPUT
none ;
BEGIN

FILENAME=`basename $thefile`

echo
echo Viewing source file in $FILENAME ...

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

RETURN "0";
END

#-----------------------------------------------------------------------
```

```
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# viewHist envelope: Views the History for an object.
#

ENVELOPE viewHist;

SHELL ksh;

INPUT
text : history;
OUTPUT
none ;

BEGIN

echo "!===================== history ======================"
$PAGER $history

RETURN "0";
END

#-----------------------------------------------------------------------
#
#                     Marvel Software Development Environment
#
#                                  Copyright 1991
#                       The Trustees of Columbia University
#                              in the City of New York
#                                All Rights Reserved
#
# yacc_compile envelope
#
# usage: compile [YFILE.contents] [YFILE.object_code] [HFILE.contents ...]
#  [YFILE.history] {LITERAL}
#

ENVELOPE yacc_compile;
```

```
SHELL sh;

INPUT
          text      : thefile;
          binary    : obj_file;
          text      : history;
          text      : ytabh;
   set_of INC       : ifiles;
       literal      : CCFLAGS;
OUTPUT
none ;

BEGIN

shortname=`basename $thefile`
dirname=`get_dirname $thefile`
log=$history

echo "running yacc on $shortname on `date`"
echo "running yacc on $shortname on `date`" >> $log
echo
echo >> $log

tmp_dir=/tmp/compile$$
mkdir $tmp_dir

# we need to make the -I list
idir=""
if [ "x$ifiles" != "x" ]
then
    ln -s $ifiles $tmp_dir
    idir="-I$tmp_dir"
fi


echo "yacc -d $shortname"
echo "yacc -d $shortname" >> $log
yacc -d $thefile
yacc_status=$?


echo "cc -c $CCFLAGS y.tab.c $idir -o $shortname"
echo "cc -c $CCFLAGS y.tab.c $idir -o $shortname" >> $log
cc -c $CCFLAGS y.tab.c $idir -o $obj_file
cc_status=$?
```

```
echo "rm -f y.tab.c"
echo "rm -f y.tab.c" >> $log
rm -f y.tab.c

# Save the y.tab.h file for later use
# ------------------------------------
echo "mv y.tab.h `basename $ytabh`"
echo "mv y.tab.h $ytabh" >> $log
mv y.tab.h $ytabh

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

RET_VAL=1
if [ $yacc_status -eq 0 ]
then
    if [ $cc_status -eq 0 ]
    then
       echo yacc successful, results available with viewHist
       echo yacc successful >> $log
       RET_VAL=0
    fi
else
    echo yacc failed, results available with viewHist
    echo yacc failed >> $log
    RET_VAL=1
fi

RETURN "$RET_VAL";
END
```

C Porting an Objectbase across different architectures

Marvel can run on different architectures, and there is no restriction on mixing and matching server with clients. However, once an objectbase is started with a specific architecture, it cannot be used by other servers with different architecture. (e.g., a Sun machine cannot run as a server on an objectbase that was initiated by a Dec station), due to binary incompatibility. In order to resolve this problem, Marvel provides two complementary external utilities: `bin2ascii` and `ascii2bin`. `bin2ascii` transforms the objectbase from binary to ASCII representation, and `ascii2bin` does the opposite.

The procedure for changing the objectbase from one (the source) architecture to another (the target) architecture, is as follows:

1. run `bin2ascii` on the source machine. This will produce an ASCII file with a representation of the objectbase.
   Invocation:
   `bin2ascii [-i infile] [-o outfile]`
   where:
   - `-i infile` specify input filename (default is data/objectbase)
   - `-o outfile` specify output filename (default is data/objectbase.ascii)
2. run `ascii2bin` on the target machine. This will produce the binary format of the objectbase that is acceptable by Marvel .
   Invocation:
   `ascii2bin [-i infile] [-o outfile]`
   where:
   - `-i infile` specify input filename (default is data/objectbase.ascii)
   - `-o outfile` specify output filename (default is data/objectbase)

Note, that if the strategy file (which contains the data-model) is changed, then both utilities will not work properly, as they interpret the data based on the data-model (the *schema*).

References

[1] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, 1991.

[2] Naser S. Barghouti and Gail E. Kaiser. Implementation of a knowledge-based programming environment. In *21st Annual Hawaii International Conference on System Sciences*, volume II, pages 54–63, Kona HI, January 1988. IEEE Computer Society.

[3] Naser S. Barghouti and Gail E. Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990.

[4] Naser S. Barghouti and Gail E. Kaiser. Multi-agent rule-based software development environments. In *5th Annual Knowledge-Based Software Assistant Conference*, pages 375–387, Syracuse NY, September 1990.

[5] Israel Z. Ben-Shaul. An object management system for multi-user programming environments. Master's thesis, Columbia University, April 1991.

[6] Naser S. Barghouti Gail E. Kaiser and Michael H. Sokolsky. Experience with process modeling in the marvel software development environment kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[7] Peter H. Feiler Gail E. Kaiser and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[8] Peter H. Feiler Gail E. Kaiser, Naser S. Barghouti and Robert W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[9] Naser S. Barghouti George T. Heineman, Gail E. Kaiser and Israel Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. In *6th Knowledge-Based Software Engineering Conference*, pages 276–287, Syracuse NY, September 1991. Rome Laboratory.

[10] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In *1st International Conference on the Software Process*, Los Angeles CA, October 1991. In press. Available as Columbia University Department of Computer Science CUCS-014-91, April 1991.

[11] Gail E. Kaiser Israel Z. Ben-Shaul and Naser S. Barghouti. An object-oriented framework for rule-based development environments. In *ECOOP/OOPSLA '90 Workshop on Object-Oriented Program Development Environments*, Ottawa, Canada, October 1990. Position paper.

[12] Gail E. Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *9th International Conference on Software Engineering*, pages 180–188, Monterey CA, March 1987. IEEE Computer Society.

[13] Michael H. Sokolsky and Gail E. Kaiser. A framework for immigrating existing software into new software development environments. *Software Engineering Journal*, 1991. In press. Available as Columbia University Department of Computer Science CUCS-027-90, May 1990.