Modules: Providing a Flexible User Environment

John L. Furlani

June 29, 1991

ABSTRACT

Typically users initialize their environment when they log in by setting environment information for every application they will reference during the session. The Modules package is a database and set of scripts that simplify shell initialization and lets users easily modify their environment during the session.

The Modules package lessens the burden of UNIX environment maintenance while providing a mechanism for the dynamic manipulation of application environment changes as single entities. Users not familiar with the UNIX environment benefit most from the single command interface. The Module package assists system administrators with the documentation and dissemination of information about new and changing applications.

This paper describes the motivations and concepts behind the Modules package design and implementation. It discusses the problems with modifying the traditional user environment and how the Modules package provides a solution to these problems. Both the user's and the system administrator's viewpoint are described. This paper also presents the reader with a partial implementation of the Modules package. Sample C Shell and Bourne Shell scripts with explanations are used to describe the implementation. Finally, an example login session contrasts the traditional user's environment with one that uses the Modules package.

Introduction

Typically, when users invoke a shell, the environment is initialized with the settings for every application they might access during a login session. This information is stored in a set of initialization files in each user's home directory. Over time, these files can incur numerous and relatively complex changes as applications move and new applications become available. Since each user has his own initialization files, keeping these files current with system-wide application changes becomes difficult for both the user and the system administrator. In this model, the user often makes environment changes during a login session by modifying the initialization files and then re-initializing the shell. The Modules package provides a way to simplify this process.[1]

The Modules package is a set of scripts and information files that provides a simple command interface for modifying the environment. Each module in the Modules package is a file containing the information needed to initialize the environment for a particular application (or any environment information). From a user's perspective, the package supplies a single command with multiple arguments that provides for the addition, change, and removal

1. Capitalized *Modules* refers to the package as a whole. References to a *module* file itself are uncapitalized. References to the *module(1)* command use italics with the man page reference.

of application environment information. From the administrator's point of view, the environment information is documented and maintained in one location with each module encapsulating one application's information. Thus, it is easier for the system administrator to add new applications and ensure that the necessary environment for the application is correctly installed and maintained by the end users.

The Design of the Modules Package

The first section describes the motivations driving the design of the Modules package. The second section presents an overview of the design.

Design Motivations

Help alleviate the burden of UNIX environment maintenance for users.

The UNIX environment is cumbersome for even experienced users. Users not familiar with UNIX are both baffled and troubled by the complexity of environment maintenance. The Modules package attempts to ease this maintenance by encapsulating environment information and providing a single command for environment modification.

Ease the dissemination of information and documentation of new software.

The amount of information an administrator must convey about a new application can be large. Many variables may need to be changed for each new application. Because each module is self-documenting, the information is readily available for reference by users.

Make it simple to change the environment numerous times during the login session.

The amount of effort involved with adding environment information as applications are needed stops users from managing their environment dynamically. Using a new application requires looking up the necessary environment modifications, making the appropriate modifications, and finally invoking the command. Thus, it is usually worth initializing the environment with the information for every application when the shell is invoked. But if adding the environment for a new application is as simple as typing a single command, the balance turns in favor of adding to the environment just before accessing a new application.

Decrease dependency on servers when the applications on those servers are not needed.

When a directory is added to the search path, a dependency on the server containing this directory is added as well. If this server goes down, the user must wait for the server to return even though, in the case of an unnecessary directory, he doesn't need access to the directory to complete his work.

Manage the difficulties associated with frequently switching between different application releases.

Switching between two releases of the same software is simplified by making it easier to swap the two applications' environment information.

Design Overview

The Modules package should be simple to use. Therefore, it employs a single command interface, very similar to that of the *sccs(1)*[1] command. A single command with the ability to provide help is both easy to remember and simple to use.

For the Modules package to assist a system administrator, it should save time spent maintaining environments as well as installing and documenting the use of new applications.

The package must be flexible enough to accommodate any situation an application might require. It must meet the user's needs in different ways. Some users will use the *module(1)* command to manipulate most of their environment. Other users will use it sparingly as an easy way to try new or rarely used applications. In addition, the package should permit experienced users to tailor it to their needs.

Finally, the solution must be shell-independent. The interface should be the same regardless of which shell the user chooses.

Problems with Traditional Shell Initialization

This section describes some of the difficulties with traditional shell initialization as viewed by the user and by the system administrator.

The User's Viewpoint

Maintaining shell start-up files can be difficult and frustrating for the user because he lacks interest or UNIX® knowledge. Since modification is generally required when a new application is installed, maintaining start-up files can be demanding and time consuming in very dynamic or large UNIX® environments.

When shell start-up files are used, the environment can become cluttered with unnecessary information. Information for every application, whether or not it will actually be referenced during the current session, is loaded into the shell. For the user to execute applications at the command line without using full pathnames, the search path could be very long.

When using an automounter[2], the system not only searches a longer path, but must remount infrequently referenced directories to search them. In addition, if a directory in the search path is on a server that goes down, the shell will hang trying to search that directory. Thus, the time needed to detect a "command not found" or to find programs toward the end of the search path is drastically increased. Shells that have path caching help this problem immensely, but a number of shells and users do not have or use path caching. It is best if a small path containing only the most used directories is set at initialization and supplemented just before a new application is used.

Switching between different versions of an application is usually difficult "on the fly." First, the user must know which environment variables to reset and then must enter the explicit shell commands to change the variables. Finally, the user must modify the search path to remove the old path and to add the new path. This is a cumbersome and time consuming process that restricts the flexibility of changing between different software versions.

Accessing a new application is difficult when it requires a change in the user's environment. If the application is for temporary use, the user accesses the application by changing the environment in the current shell. If it's a long-term addition to the user's set of applications, the user edits the shell initialization files and the shell is re-initialized. Novice users often don't know how or don't want to know how to modify their environment to use the new application.

The System Administrator's Viewpoint

A system administrator currently announces the installation of a new application via e-mail or a note in /etc/motd. Usually, the notice contains a full description of the application and the environment variables that must be set to use the new application. Some users do not understand what they really need to do to use an application. This, in turn, causes numerous requests to a system administrator. Novice users often need a system administrator to help them modify their start-up files.

At most sites, a logfile or database is maintained containing the descriptions and quirks of each application so that the user can set up his environment to use an application. Maintaining such a logfile can be time consuming as it can become very large.

The Modules Package Provides a Solution

The User's Viewpoint

Although shell start-up files are still necessary, maintaining them is easier with the Modules package. The user is provided with two options: modify the start-up files directly, or use the *module(1)* command to modify them.

Changing start-up files is simpler with the Modules package. The user only has to add new arguments to, or remove them from, the *module(1)* command in his start-up file to add or remove an application's environment. Or, if the user prefers, he can use the *module(1)* command to add module names to, or remove such names from, his start-up file's *module(1)* command.

A clean environment is readily maintained since the Modules package makes it easy to dynamically modify the environment. A minimum of environment information is initialized at start-up, and an application's environment is added only when needed. Response time is improved because search paths are much shorter on average.

The Modules package is optimum for the windowing environment under which many UNIX users work. For example, a user only loads the window system module during the login initialization. Then, in a shell window, the user uses the *module(1)* command to initialize the environment just prior to accessing a new application.

If the path to an application changes, the change will be masked by the Modules package. For example, a user loads a module named 'openwin' to use OpenWindows[3] even if its access path has changed. For the user, no environment or start-up file modification is required.

To switch between different releases, the user simply changes predefined modules. The old module is swapped out, and the new one is loaded in its place, even during the login session.

The Modules package will help inexperienced UNIX users manage their environment. They must learn a single command for manipulating their environment. This is opposed to having to thoroughly understand the quirks of setting a UNIX environment.

The System Administrator's Viewpoint

The Modules package eases the dissemination of information about a newly installed application: only the module name must be announced. If users want to use the new application, they use the *module(1)* command to add the announced module. Any users who don't use the Modules package can acquire the information needed to set their environment from the module file itself.

Each module is self-documenting. Users either access this information via the *module(1)* display command or view the module file itself. In general, a logfile or database still needs to be maintained, but only the module name is listed for each application. Thus, when a user wants to use an application, the database references a module name that contains the current environment information. The user either loads this module directly or gets the environment information from the module and manually incorporates it into the environment.

Shell Wrappers and Modules

Shell wrapper scripts setup environment variables for a certain application when a command for that application is invoked. For each application, the system administrator creates a wrapper script and a symbolic link to the script for each command in the application. The Modules package can augment a wrapper script scheme or be used in place of wrapper scripts.

With wrapper scripts, users still add the directory containing the symbolic links to their search path in order to use the application. In this case, the Modules package augments the wrapper scheme by helping the user manage the search path.

One solution for managing the search path is creating a directory of symbolic links to all of the wrapper scripts. In this case, the user only adds one directory to his search path to access every application. Moving an application requires that every symbolic link for that application change. When many applications are installed, this directory can quickly become overwhelming and unmanageable. Documenting and finding programs in such a directory is difficult and often not very clean.

The Modules package provides the user with a lot of flexibility by differentiating between user and system module files. Like wrapper scripts, the

Modules package can encapsulate environment details from the user.

The Modules Package Implementation

The Modules package has been implemented for both the C Shell and Bourne Shell dialects. This section describes some of the implementation details.

Modules Initialization

The Modules package is initialized when a user sources a site-wide accessible initialization script. This file is shell dependent and is usually done in a user's *.cshrc* or *.profile* upon each shell invocation.

This Modules initialization script defines a few environment variables. MODULESHOME is the directory containing the master module file and the master command scripts. MODULEPATH is a standard path variable that is searched to find module files. MODULESHOME should always be a part of this path. The *_loaded_modules* variable contains a space separated list of every module that has been loaded. All of these variables are exported so that the shell's children will have the same information and be able to keep track of currently loaded modules.

This initialization script sets up the *module(1)* command. This command is an alias or a function depending upon which the shell supports (see Figure 1).

FIGURE 1. *module(1)* Command Initialization

```
##
##  module(1) User Command as Function
##
module() {
 _module_argv="$*"
 . $MODULESHOME/.module.sh
 unset _module_argv
}
```

Environment Modification

Because a process is unable to modify the environment of its parent, the Modules package sources scripts into the current shell.

The Modules package should not alter any part of the environment besides the variables documented in the *init-script* or the module files themselves. When a script is sourced into the current shell, it has the potential to change existing user variables. This "feature" permits the Modules package to work, but it presents a pitfall that the package must take into account. The main concern is the *module(1)* command because it uses a large number of variables to implement its sub-commands.

A couple of precautions have been taken to avert the possibility of variables being changed or destroyed by the *module(1)* command. The first precaution is the choice of variable names used by the *module(1)* command. All of the names are proceeded with an underscore and are all lowercase. The use of underscores should stop most variable conflicts. It does, however, leave room for a user variable to be changed by the *module(1)* command. So, a check for possible variable conflicts is made when the bulk of the script is sourced. If a conflict arises, the user is notified.

The problem of changing existing variables could be eliminated by running *module(1)* as a subshell and sourcing the return values. I found this has an unacceptable response time for the problem being addressed. If users run into variable conflicts, they can set an option telling the *module(1)* command to run the script in a subshell and source the script's output (this code is not in Figure 1).

Since the Modules package is designed to abstract environment information from the user, it must be concerned with environment dependencies and conflicts between different applications and different versions of the same application. For example, the environment for two versions of the same application should not be loaded at the same time. Along the same lines, some applications (like Sun's AnswerBook[4]) are dependent upon other applications (OpenWindows[3]). Possible conflicts and dependencies are put into the module files themselves and detected by the *module(1)* command.

Shell Independence

The Modules package is shell-independent because the interface is independent of the currently executing shell. To ease administration, the module files are shell independent as well. Thus, only one copy of an application's environment information is maintained.

A number of functions or aliases are set up by the *module(1)* command. Each module calls these functions to accomplish specific, well defined tasks. For example, the *_set_environ* task is responsible for setting an environment variable. This is a line from an "openwin" module file.

```
_set_environ   OPENWINHOME    /depot/openwin
```

Here, OPENWINHOME is initialized to /depot/openwin[5].

System Module Files and User Module Files

Through the MODULEPATH environment variable, users can specify module directories to be searched before or after the site-wide directory. Thus, a distinction is made between "system modules" and "user modules."

System modules are maintained by the system administrator and contain the default initialization information for packages that are installed on the network. User modules, created by the user, are either derivatives of the site-wide modules or are new modules that are specific to the user's needs. This arrangement provides the user with the same power as the system-wide modules provide to easily change the environment "on the fly."

Implementation of Internal Module File Functions

This is the set of internal functions called by the module files themselves. These functions change paths and set variables, aliases, and dependencies.

_prepend_*path and _append_*path

Sometimes an application's path should be prepended to a search path. Other times it should be appended to a search path. The Modules package makes this distinction in the application's module file. The path modification functions are currently implemented to modify the PATH, MANPATH, MODULEPATH, and LD_LIBRARY_PATH environment variables. *See Figure 2 for an example of how the alias and function to append the MANPATH variable is implemented.*

FIGURE 2. *_append_manpath* Alias and Function

Bourne Shell Function:

```
_append_manpath() {
if [ "$_rm_flag:-X" = "X" ]; then
    _rm_manpath $1
else
    MANPATH="$MANPATH":"$1";
    export MANPATH;
fi
}
```

C Shell Alias:

```
alias _append_manpath
    'if($?_rm_flag) eval _rm_manpath \!:1;
    if(! $?_rm_flag) setenv MANPATH
    $MANPATH":"\!:1'
```

_rm_*path

The *_rm_path* functions remove the directory, given as an argument, from their associated path. As with the append and prepend path functions, they're currently defined for the PATH, MANPATH, MODULEPATH, and LD_LIBRARY_PATH environment variables. Currently, *awk(1)*[?] does most of the work of removing and recreating the path.

When the user requests that a module be removed, the *_rm_flag* is set in a higher level function. Then, the module is reloaded with this flag set. Thus, every function that was called when the module was loaded is called again with the *_rm_flag* set (see Figure 2). The same functions that set up the environment now call their associated *_rm_*path* function to remove their environment information.

See Figure 3 for an example of how the function to remove a directory from the MANPATH variable is implemented.

FIGURE 3. *_rm_manpath* Function and *awk* Script

As Bourne Shell Function:

```
_rm_manpath() {
    MANPATH=`echo $MANPATH |
        awk -F: 'BEGIN {p=0}
        {
            for(i=1;i<=NF;i++)
            {
                if($i!="'$1'"){
                    if(p) {printf ":"}
                    p = 1;
                    printf "%s", $i
                }
            }
        }'`
  export MANPATH;
}
```

_set_environ

This function is responsible for setting and clearing environment variables. The code gets more complex when removing environment variables.

Since environment variables are often used to define paths to other directories, variables and paths defined later in a module must be able to reference these variables even as they are removed. The best way to describe this problem is with an example module file (see Figure 4).

FIGURE 4. OpenWindows module File

```
##
##  OpenWindows Version 2.0
##
_set_environ OPENWINHOME   /depot/openwin
_set_environ DISPLAY       `hostname`:0.0
_prepend_newpath   $OPENWINHOME/bin/xview
_prepend_newpath   $OPENWINHOME/bin
_prepend_manpath   $OPENWINHOME/man
_prepend_ldpath    $OPENWINHOME/lib
```

Here the first variable set is the location of OpenWindows (OPENWINHOME). This variable is used to define the path locations as well. If the variable were removed from the environment before the paths were removed, it would be impossible to remove the paths. So, environment variables are not actually cleared from the environment until after the module has been removed. Thus, *_set_environ* simply adds any environment variables to a list. Upon completion of reading the module file, the elements are removed from the environment. *See Figure 5 for an example of how the alias and function defined to set and unset environment variables are implemented.*

FIGURE 5. *_set_environ* Function and Alias

Bourne Shell Function

```
_set_environ() {
 if [ "$_rm_flag:-X" = "X" ]; then
 _unset_list="$_unset_list $1"
 else
 eval $1="$2"; export $1;
 fi
}
```

C Shell Alias

```
alias _set_environ '
    if($?_rm_flag)
        set _unsetenv_list =
            ($_unsetenv_list \!:1);
    if(! $?_rm_flag) setenv \!:1 \!:2'
```

_prereq and _conflict

Two functions were created to manage conflicts and dependencies with other module files. The *_prereq* function is a list of modules the calling module must have loaded to run. Similarly, the *_conflict* function is a list of modules the calling module has conflicts with. If more than one module is listed for these commands, it is treated as an ORed list. Multiple calls can be used to get an ANDing effect.

For example, AnswerBook needs OpenWindows loaded to run. So, it defines the openwin (or openwin-v3) module as a prerequisite. A conflicting case would be OpenWindows Version

2.0 with OpenWindows Version 3.0. These two modules should not be loaded at the same time. So, each one defines the other as a conflict. *See Figure 6 for an example of how the function for conflict management is defined.*

FIGURE 6. *_conflict* Function

```
_conflict() {
    if [ "$_rm_flag:-X" = "X"]; then
        return;
    fi

    _conflict_loaded=0
    for _con in $*; do
        for _mod in $_loaded_modules; do
            if [ "$_mod" = "$_con" ]; then
                _conflict_loaded=$_mod
            fi
        done
    done

    if [ $_conflict_loaded != 0 ]; then
        echo "ERROR: Module conflict"
    fi
    unset _conflict_loaded
}
```

Implementation of the *module(1)* Command

Each invocation of *module(1)* sources a site-wide script that actually implements the command. Arguments are passed to the *module(1)* command using the *_module_argv* variable (see Figure 1).

The first argument designates the sub-command the *module(1)* command is to execute. Valid arguments are as follows:

- Load or Add
- Remove or Erase
- Switch or Change
- Show or Display
- Initadd
- Initrm
- List
- Available
- Help

Loading Modules

Loading modules is done by the *_add_module* internal function. This function takes any number of arguments and attempts to load each one as a module. It traverses the argument list first verifying that a listed module isn't loaded already. If it is, the function prints an error and moves on to the next name in the argument list (see Figure 7).

FIGURE 7. *_add_module()* Argument Verification

```
_add_module() {
if [ $# -lt 1 ]; then
    echo "ERROR: More arguments"
    return
fi

for mod in $*; do
    _found=0; _cur_module=$mod;
    for chkmod in $_loaded_modules; do
    if [ $mod = $chkmod ]; then
        _found=1
        break
    fi
done

if [ $_found -eq 1 ]; then
    echo "ERROR: Module is already loaded"
    continue
fi
```

If the module is not loaded, *_add_module* begins looking for the module by searching each directory specified in the *MODULEPATH* variable. Once found, the module is sourced, the module name is appended to the *_loaded_modules* variable, and if there are any other arguments, the load process begins anew (see Figure 8).

FIGURE 8. *_add_module()* Traverse *MODULEPATH* and Load Module

```
    for dir in $MODULEPATH; do
        if [ -f $dir/$mod ]; then
            echo "Loading $dir/$mod"
            . $dir/$mod

            if [ "${_load_error:-NotSet" =
                        "NotSet" ]; then
                _loaded_modules=
                    "$_loaded_modules $mod"
                export _loaded_modules
                unset _load_error
            fi

            _found=1
            break
            fi
    done
    if [ $_found -ne 1 ]; then
        echo "ERROR: Module not found"
    fi
done
}
```

Removing Modules

Removing a module is accomplished by using the *_rm_module* internal function, which is very similar to the *_add_module* function. Any number of arguments can be passed to the *_rm_module* function. First, each argument is checked to verify that the module is actually loaded (same code as in Figure 7 except the final *if* statement indicates an error if the module is not loaded). If a module is loaded, the *_rm_flag* is set and the *MODULEPATH* variable is searched. Once a module is found, it is sourced (see Figure 9).

FIGURE 9. *_rm_module()* Traverse *MODULEPATH* and Load Module

```
    _rm_flag=
    for dir in $MODULEPATH; do
        if [ -f $dir/$mod ]; then
            echo "Removing $dir/$mod"
            . $dir/$mod

            _loaded_modules=
    `echo $_loaded_modules | sed s/$mod//`

            export _loaded_modules
            _found=1
            break
        fi
    done

    if [ $_found -ne 1 ]; then
        echo "ERROR: Module not found"
    fi

    for env in $_unset_list; do
        unset $env
    done
```

The same functions that are used in loading a module are used to remove modules. When one of the functions detects that the *_rm_flag* is set, it removes its corresponding piece of environment instead of adding it (see Figures 2 and 3). Note that after the module file has been sourced, each variable in the *_unset_list* is unset.

Switching Modules

Although this function has not been fully implemented, I will describe it here. Only modules that define themselves as compatible with another module can be switched. The compatibility information is kept in the module file. If a module can be switched with another module, it lists that other module via the *_switch* function.

Switchable modules are very similar in that their environments match one another and their modules follow the same format. The variables that have to be reset are the same, and the search path changes are the same as well.

The difference between switching two modules and the process of removing a loaded one and loading a new one is that the location of a search path entry does not change. The append and prepend functions are used when removing and loading module files. This process has the possibility of altering a portion of the search path in relation to other entries.

Although this sounds restrictive, it is often very useful because most module switching involves different versions of the same program.

Displaying Modules

The *_display_module* internal function implements the user display and show sub-commands. If no arguments are provided, it displays information about every loaded module. Otherwise, only the modules named as arguments are displayed. An *awk(1)* script is used to convert the information contained in the module file to output that is visually pleasing to the user.

Changing User Initialization Files

The *initadd* and *initrm* sub-commands help the user add modules to and remove modules from their shell initialization files. The initialization file is searched for a comment line placed there by the *module(1)* command. Located immediately after this comment line is a line invoking the *module(1)* command. It is this line that is changed according to the list of modules given to the *initadd* and *initrm* request. If a comment line is not located in the initialization file and the user is requesting that a module be added, the comment line and the *module(1)* command line is appended to the user's initialization file.

Listing Modules

The *_list_modules* function simply prints out the current value of the *_loaded_modules* variable.

Available Modules

Each directory in the *MODULEPATH* variable is listed using the UNIX *ls(1)*[?] command by the *_avail_modules* function (see Figure 10).

FIGURE 10. *_list_modules()* and *_avail_modules()* Functions

```
_list_modules() {
    if [ "$_loaded_modules" ="" ]; then
        echo "No Modules Loaded"
    else
        echo "Loaded: $_loaded_modules"
    fi
}

_avail_modules() {
    for dir in $MODULEPATH; do
        echo $dir":"
        (cd $dir; ls)
    done
}
```

Help for Modules

Two levels of help are provided. Without any arguments, the *module(1)* command lists the available sub-commands. With 'help' as the only argument, it provides more complete description of the Modules package. If a second argument, the name of a sub-command, is provided, the *module(1)* command displays help about the sub-command.

Example Sessions

Contrast Conventional Style with Modules Style

The examples in Figures 11 and 12 depict how the same environment modifications are accomplished with and without using the Modules package. Specifically, the examples show how a user would switch from using OpenWindows Version 2.0 to a Development Version of OpenWindows Version 3.0. The keystrokes the user actually types are in bold. Notice the difference in effort needed to switch between the two window systems. Also notice the difference in the length of the search path variables.

FIGURE 11. Conventional Style

```
system login: jlf
++++++++ CSH Login ++++++++
jlf@system% echo $PATH
/depot/lang:/depot/openwin/bin:
/depot/openwin/bin/xview:/usr/local/bin:
/usr/bin:/usr/ucb:/usr/etc:.:
/depot/frame/bin:/depot/sunvision/bin:
/depot/TeX/bin
jlf@system% setenv OPENWINHOME /depot/openwin-v3
jlf@system% setenv PATH /depot/lang:
/depot/openwin-v3/bin:/depot/openwin-v3/bin/xview:
/usr/local/bin:/usr/bin:/usr/ucb:/usr/etc:.:
/depot/frame/bin:/depot/sunvision/bin:
/depot/TeX/bin
jlf@system% setenv LD_LIBRARY_PATH
/depot/lang/SC1.0:/depot/openwin-v3/lib:/usr/lib
jlf@system% setenv MANPATH /depot/lang/man:
/depot/openwin-v3/man:/depot/sunvision/man:
/depot/TeX/man
jlf@system% openwin
```

FIGURE 12. Modules Style

```
system login: jlf
++++++++ CSH Login ++++++++
Loading /site/Modules/openwin
jlf@system% echo $PATH
/depot/openwin/bin:/depot/openwin/bin/xview:
/usr/local/bin:/usr/bin:/usr/ucb:/usr/etc:.
jlf@system% module rm openwin
Removing /site/Modules/openwin
jlf@system% module add openwin-v3
Loading /site/Modules/openwin-v3
jlf@system% openwin
```

A More Complex Modules Example

A few more *module(1)* commands are demonstrated in Figure 13. Once the user logs in, a check is made of what modules are currently available. Then, the 'lang' module is displayed to find out what the module does. Notice that the PATH environment variable is changed as the module display indicates. The 'answerbook' module is displayed showing how prerequisites might be used. In this example, answerbook must have either the 'openwin' or 'openwin-v3' module loaded before it will load. Finally, the 'answerbook' module is loaded and the program is started.

FIGURE 13. Complex Modules Example

```
system login: jlf
++++++++ CSH Login ++++++++
Loading /site/Modules/openwin
jlf@system% module avail
/site/Modules:
X11@       init-csh   openwin         tex
X11R4      init-sh    openwin-v3      vx-devel
answerbook lang       saber           xgl
dos        local      sunvision
frame      lotus      sunvision-devel
frame-ol   mh         taac-devel
jlf@system% module show lang
++++++++ ( /site/Modules/lang Module ) ++++++++
Unbundled Languages
Prepend PATH: /depot/lang
Prepend MANPATH: /depot/lang/man
Prepend LD_LIBRARY_PATH: /depot/lang/SC1.0
jlf@system% echo $PATH
/depot/openwin/bin:/depot/openwin/bin/xview:
/usr/local/bin:/usr/bin:/usr/ucb:/usr/etc:.
jlf@system% module add lang
Loading /site/Modules/lang
jlf@system% echo $PATH
/depot/lang:/depot/openwin/bin:
/depot/openwin/bin/xview:/usr/local/bin:
/usr/bin:/usr/ucb:/usr/etc:.
jlf@system% module show answerbook
+++++ ( /site/Modules/answerbook Module ) +++++
Answerbook Version 1.0
Prerequisites(ORed): openwin openwin-v3
Append PATH: /depot/answerbook
jlf@system% module add answerbook
Loading /site/Modules/answerbook
jlf@system% answerbook
```

Future Work

Having the Modules package as a set of scripts that are sourced into an existing shell helps makes the interface shell independent. However, it would be best for performance and cleanliness to have support for the Modules package built into the shell itself. The module commands could be more complex without losing any performance over the current version.

Currently, conventional style search paths can be built in an order that doesn't represent a logical search structure. Users are responsible for maintaining the order of their paths with little or no help. The Modules package can provide the user with the information he needs to build a logical search path with existing modules.

Work is in progress to increase the grammar available in the module file. Syntax permitting if-else statements and path ordering are two examples of new syntax not described in this paper.

Finally, more options are under development to provide users with even greater control over how their environment is constructed by the Modules package. In some cases, users don't want a variable modified when loading a certain module file. For example, if the OpenWindows[3] dynamic libraries are already cached using *ldconfig(8)*[1] the LD_LIBRARY_PATH should not be modified when loading the "openwin" module file. Other configuration and control options are being added for more experienced users as well.

Results, Performance Notes

The Modules package is quite new and is still under development. Only a few of our users have begun working in the Modules' environment. They have been very pleased with the package and the benefits it provides.

Currently, it takes a second or two to load a module into the current shell. In an effort to improve this performance, it is possible for the user to specify that the internal functions remain resident from one *module(1)* command to the next. This provides a marked improvement in speed since the functions are not being completely redefined upon every invocation.

Summary

The Modules package provides both the novice and the experienced UNIX user with a clean interface to the environment. This interface enables the user to easily add, change, and remove application environments dynamically.

John L. Furlani graduated from the University of South Carolina with a BS in Electrical and Computer Engineering. he worked as a system administrator at both USC and the Naval Research Laboratory in Washington, D.C. during his college years. Upon graduation, John joined Sun Microsystems Incorporated as the system administrator for Sun's Research Triangle Park Facility in North Carolina. Reach him at Sun via U.S. Mail at Sun Microsystems Inc., P.O. Box 13447, Research Triangle Park, NC 27709-13447. Reach him via electronic mail at sun!sunpix!jlf@uunet.uu.net or via the internet at John.Furlani@East.Sun.COM

Acknowledgments

Ken Manheimer and Don Libes at the National Institute of Standards and Technology deserve special thanks for their help and ideas toward this paper and some design considerations. Maureen Chew with Sun Microsystems provided me with a test environment and many ideas on how to improve Modules. There are many others that deserve thanks but too many to list here -- thanks to everyone that helped.

References

[1] Sun Microsystems Incorporated, *SunOS Reference Manual.*

[2] Sun Microsystems Incorporated, "Using the NFS Automounter", *System and Network Administration*, Chapter 15.

[3] Sun Microsystems Incorporated, *OpenWindows Version 2 Reference Manual.*

[4] Sun Microsystems Incorporated, *Using the Sun System Software Answerbook.*

[5] Manheimer, Warsaw, Clark, Rowe, "The Depot: A Framework for Sharing Software Installation Across Organization and UNIX Platform Boundaries", *USENIX Large Installation System Administration IV Conference Proceedings,* October 1990, p. 37-76.