Drag your design environment kicking and screaming into the '90s with Modules!

Erich Whitney, Axiowave Networks
Mark Sprague, ATI Research

ewhitney@axiowave.com
msprague@ati.com

ABSTRACT

Creating and maintaining a design environment is a task that few people enjoy doing, yet it is a critical part of any design team's success. Every tool vendor creates some sort of installation environment for their own tools, but integrating tools from multiple vendors can be a challenge. This paper presents some tips and tricks to building a flexible design environment that is easy to use, understand, extend and maintain.

At the core of this environment is the public-domain Unix package "Modules" that provides a modular utility for managing tool installations. Modules can also be applied to managing project environments, and user environments as well. Modules uses simple scripts written in a Tcl superset called "modulefiles" to create an environment. A single Unix command is used to load and unload environments for different tools, check for tool and platform dependencies, grab the correct version of a tool, etc. This eliminates the problems such as "Path too long", running out of environment space, using an incorrect version of a tool and many others. Since all the tool setup is done in Tcl, users can choose any login shell they like and they can share modulefiles because they're in a common, standard language.

This paper shows how a design environment is constructed around Modules and demonstrates examples of using Synopsys' Design Compiler and VCS with LSF in a non-homogeneous compute environment.

1.0 Introduction – So what's the problem with our design environments anyway?

The typical hardware designer needs frequent access to a large number of tools. The tools might include, among others, simulation, synthesis, data path, static timing, editors, and waveform viewers. If the designer is also performing physical design, the list of tools required can easily double, with floor planning, placement, clock tree insertion, detailed routing, and parasitic extraction.

If you are working on isolating a bug, qualifying a new release, or trying to get 'optimal' results from many tools, you might require simultaneous access to several versions of various tools. How often have you heard a vendor suggest that in order to get around a problem, you use the following sequence:

1. Analyze/elaborate your design using version A.
2. Compile your design using version B.
3. Generate your reports using version A.

Or have you ever had a designer maintain that the only way to get the results they need is to use version X of a tool, when the rest of the project is using version X+4?

Finally, try to transparently incorporate a load sharing tool, on a non-homogeneous compute environment, and you can see why most CAD departments are a bit gruff. It also explains why there is a great deal of resistance to adding support for the latest hot new operating system – Linux, NT, AIX, HPUX…

So how can you build a tool environment that will solve the following problems:

1. Provide access to multiple versions of each tool.
2. Provide a mechanism to easily switch between tool versions.
3. Limit the overhead required by CAD/IT to install a new version of a tool.
4. Minimize effort to add CAD tools to the environment.
5. Allow project specific setup of the CAD tools.
6. Support multiple Unix shells.
7. Support a heterogeneous compute environment.
8. Provide a standard mechanism for adding tools to a users environment.

1.1 How is the problem being solved?

Different portions of the wish list have been solved using a variety of mechanisms. We'll outline a few of the ones we have run into.

1.1.1 Go it alone

In some instances, the IT department washes its hands of any support for the CAD environment. This certainly limits the overhead required by IT to install and support new tool versions. However, it fails miserably on all other scores.

1.1.2 Homogeneous Environment

Some CAD/IT departments use the Henry Ford approach to solve the problems of supporting multiple Unix shells and a heterogeneous compute environment: Any color as long as it's black. In this case, CAD/IT dictate a standard Unix shell and settle on a single compute platform. This simplifies development of a standard login script and eliminates the need to handle multiple executables.

Unfortunately, it also potentially locks the organization into non-state of the art processors. This approach also necessitates upgrading the entire compute farm to change platforms, instead of incrementally adding new systems.

The standardization on a single shell is not an issue for many users. There are those users that *do* have a very strong preference for a particular shell and these users will generally chafe under the single shell requirement.

1.1.3 Single login script

This usually starts out innocently when there are a few tools, and only a couple of revs. This approach allows a single script to setup all of the tools. This allows the CAD or IT groups to debug and maintain a single view of the world to the users. In this approach, there are additional shell variables used to enable or disable various tools, and also to determine the version of the tool the user wants. This approach can also provide the hooks required for support of multiple platforms.

However, there are a few shortcomings with developing and using a single login script. First, it forces all users into a single family of shells. Second, because the script is centrally maintained, it can be difficult to add experimental tools, or beta versions of existing tools. Finally, these scripts can grow to be excessively long. At one company that used this approach, the script grew over the years to:

```
# wc cshrc_sys.NEW
            7128   26760  246595 cshrc_sys.NEW
```

So this 'simple' login script has grown to over 7000 lines! Needless to say, it's maintainability is starting to suffer somewhat.

1.1.4 Individual Shell scripts per Tool

An alternative to the single login script is to break it into multiple smaller scripts, each of which sets up a single tool. This provides a set of more maintainable scripts, but does not alleviate the need to require a single family of shells to be used. It does provide the ability to add new tools into the environment easily. Multi-platform support can also be built in for tools providing this option. This approach does not provide an easy means to switch between different versions of a single tool. Also, it is up to the script writer to provide comments or something to notify the user of what versions are available and what to do if an unavailable version is requested.

1.2 So what's the answer?

Every company solves this problem in a different manner. We've presented some alternatives that address some but not all of the desired features of a tool environment. Clearly, there must be a solution that addresses all of these issues, and maybe one that has been in use for several years, so the bugs have been worked out of it. Enter Modules.

2 Introducing Modules

Modules[1] is a software utility that allows you to dynamically configure your Unix environment for any number of software packages. Modules is written in C but is configured using Tcl scripts which have extensions specifically designed to manipulate the user's environment. Modules abstracts the details about a software package's installation, environment, and software dependencies so that from a user's perspective, all software is accessed in the same way. User's no longer have to worry about keeping track of environment variables (such as PATH and MANPATH) or other installation details.

An environment built on Modules has several nice features. All applications can be setup with a standard scripting language and methodology. Users have the option to use any login shell they choose without fear of being alienated because of some arbitrary decision to use only one type of shell. The environment is more robust because the modulefiles can be programmed to accommodate your computing environment. Standards are easily defined and maintained. Also, application versions can be easily tracked, added, or removed dynamically in the environment. Modulefiles can be created to encapsulate a suite of tools for a user, a project, or a company—making the Unix environment easier to use. Furthermore, tool and version conflicts can be detected and handled automatically by the modulefile.

Modulefiles are easy to create and are typically less than 10 lines of Tcl each. Modulefiles can be built hierarchically to any number of arbitrary levels which enables code re-use between modules.

A typical EDA tool will come with an installation script that will add to the user's PATH through a setup script or by manual modification of the user's login environment. Using Modules requires a one-time modification of the user's startup file. After that, software environments can be dynamically created, removed, loaded, and unloaded with a single command. Creating modulefiles is simply a matter of taking the shell commands from the software packages setup script or documentation and converting them into Module's Tcl extensions (setenv, prepend-path, append-path, etc.)

[1] Capitalized *Modules* refers to the software package as a whole. References to a *module* file (or *modulefile*) itself are not capitalized. References to the *module(1)* command use italics with the man page reference.

3 Building an Environment with Modules

3.1 Introduction

This section will go through the process of building an environment with Modules. The steps outlined here reflect an example of how a particular design environment was built. The intent here is to lay out the key steps in using Modules so that others can adapt it to their needs.

3.2 Installing Modules

Modules is quite easy to use once you have it installed. Installation, however, can be a bit of a pain. This is where it's good to have someone around who is resourceful and has some experience with the Gnu tools. The most annoying problem we found was that the Modules package is pretty picky about your Tcl installation since Modules is completely dependent on Tcl in order to run. Our Unix guru had to perform a clean Tcl install in order to build Modules.

One other important note about installation—you must install the Modules software in a location that every user will be able to access it when they login. We chose /usr/local because for us that was an NFS-mounted directory built and attached to all machines in our environment. If you have different machines in your environment, then you'll have to make sure you build a version of Modules for every architecture but the key is to remember to make the Modules install look the same on every machine.

In our environment, Modules is installed in **`/usr/local/Modules/3.1.2`**. The user's .cshrc, .profile, or .bashrc script points to location of the installed version.

Next, we created a common login environment for everyone so that the Modules environment would be setup automatically on login. This is the one place you have to worry about handling the user's login shell preference. In the init directory, there is a shell file for each type of login shell (sh, csh, and their variants):

```
% ls /usr/local/Modules/3.1.2/init

bash  csh   ksh   perl  sh    tcsh  zsh
```

To handle the user's choice of scripts, we determined the least common denominator among all of the login files and came up with these three: The .cshrc file is used by csh, tcsh, and other csh variants, the .profile file is used by sh/zsh/ksh, and .bashrc is used by bash. Here's what these files look like:

Here is the .cshrc file (this also works for tcsh and other csh variants):

```
#
set history=128
set filec
setenv HOSTNAME `hostname`
setenv DEFAULT_MODULES "modules common perl lsf"
#
# Set mver to the desired Modules version
```

```
#
set mver=3.1.2
if ( $?tcsh ) then
  set minit=/usr/local/Modules/$mver/init/tcsh
else
  set minit=/usr/local/Modules/$mver/init/csh
endif

#
# This code should only execute if you're starting from scratch
# Otherwise, reload the environment
#
if ( -e $minit ) then
  source $minit
  if ( "$LOADEDMODULES" == "" ) then
    module use /usr/local/Modules/modulefiles/groups
    module use /usr/local/Modules/modulefiles/projects
    module use /usr/local/Modules/modulefiles/tools
    module unuse /usr/local/Modules/modulefiles
    module load ${DEFAULT_MODULES}
  else
    module update
  endif
else
  echo "Modules not supported on this system"
endif

# If the user has additional customizations

if ( -e ~/.cshrc_custom ) then
   source ~/.cshrc_custom
endif
```

See the appendix for the corresponding .profile and .bashrc files.

Each of these files do basically the same thing. First, check to make sure we are in a supported shell, then check to see if the Modules environment is already loaded (we are just a new shell). Second if Modules is not initialized, call the appropriate shell init file, load some basic modulefiles, then load a user's custom modulefile if it exists in **~/.modulefiles/*username***.

Finally, if the user wants to further customize their shell with shell-specific commands, then call the appropriate customization script in their home directory.

At this point, the user has the Modules environment. They have access to a common basic set of Unix utilities and they have optionally loaded any customizations. Putting your favorite aliases and environment settings in your own modulefile allows you to share those settings with other users without having to worry about which shell they're using.

Now let's take a look at how the modulefiles are organized. We chose to organize them into the following four categories:

```
% ls /usr/local/Modules/modulefiles
```

```
groups/   projects/ tools/
```

The /usr/local/Modules/3.1.2/modulefiles directory contains all the modulefiles that came with the Modules distribution and most importantly contains the ‘modules’ modulefile. The tools directory contains all of the modulefiles for the EDA and other software tools we use. The groups directory contains the modulefiles for the various teams (like designers, verification, synthesis, FPGAs, etc.). And the projects directory contains the modulefiles for all of the design projects that we’re managing with Modules. More on this later.

3.3 Defining a Directory Structure

Now let’s see how Modules relates to the software tools we are going to use. On our system, there is one file server for all of the software tools. Each time we install a new package, it goes in:

/tools/vendor_name/tool_name/version_number

Then, we create a corresponding modulefile in:

/usr/local/Modules/modulefiles/tools/tool_name/version_number

So the user only has to know the name of the tool and if they really care, they can specify the version number. Later on, we’ll show how even this information can be simplified with the “group” and “project” modulefiles.

3.4 Setting up a tool with Modules

In order to access a modulefile to setup part of your environment, there are a couple of files that need to be setup properly.

The first file that is required represents the version of the tool being referenced. In most cases, this should reflect the version numbering used by the tool vendor. So for VCS, these files would be named **5.2.1, 6.0**…

The second file is the **.version** file, which is used to define the default behavior when the modulefile is loaded. This file isn’t required if only one version of the tool exists, but it is good practice to create these files when the initial modulefile is created. The version names referenced here need to match the file names created above. More information on this file is given in section 3.7.

3.5 Using Modules

Users now have basically what they need to use modules. Here are some examples of the sorts of things you do with the module command.

To see what modulefiles are available:

```
% module avail

-------------------- /home/users/ewhitney/.modulefiles ----------------------
```

```
ewhitney

------------------- /usr/local/Modules/modulefiles/tools --------------------
covermeter/4.0.2(default)     lsf/4.0.1(default)
dc_shell/2000.11(default)     netscape/4.08(default)
denali/2.7                    perl/5.6.0(default)
denali/2.700_022              polaris/2000.2(default)
denali/2.9000_003(default)    primetime/2000.11(default)
dinotrace/9.1d(default)       proverilog/2.5.2
emacs/20.4                    purify/5.3(default)
emacs/20.7(default)           speedsim/3.2.0(default)
explorertl/2000.2.08(default) surelint/2.0.9(default)
forte-developer/6.0(default)  syscomp/4.0(default)
fpga_compiler2/3.5            tcl/8.2(default)
fpga_compiler2/3.5.1          timingdesigner/5.2(default)
fpga_compiler2/3.5.2          vcs/5.2.1
fpga_compiler2/3.6(default)   vcs/6.0(default)
gcc/2.95.2(default)           vcs/6.0Beta2
hdlscore/3.1.5(default)       vile/9.2(default)
hs2/2.26(default)             xilinx/3.1i
java/1.3(default)             xilinx/3.3i(default)

------------------ /usr/local/Modules/modulefiles/projects -------------------
project1     chip1   chip2

------------------ /usr/local/Modules/modulefiles/groups --------------------
common fpga   sim    syn

------------------------ /usr/local/Modules/versions ------------------------
3.1.0 3.1.2

-------------------- /usr/local/Modules/3.1.2/modulefiles --------------------
dot         module-info modules     null        use.own
```

Note: The versions followed by **(default)** are the ones specified in the corresponding **.version** file.

To load a modulefile:

```
% module load dc_shell
```

To see which modulefiles you have loaded:

```
% module list
Currently Loaded Modulefiles:
1) modules        4) tcl/8.2        7) vile/9.2       10) dc_shell/2000.11
2) common         5) perl/5.6.0     8) emacs/20.7
3) netscape/4.08  6) lsf/4.0.1      9) ewhitney
```

And to remove a modulefile from your environment:

```
% module unload dc_shell
```

You can also switch versions, reload the environment, and purge all modulefiles. See the *module(1)* man page for more information.

3.6 Designing modulefiles

The Modules package extends standard Tcl by adding some basic shell-like commands that are designed specifically to address environment variable management. These extensions include; setenv, prepend-path, append-path, and remove-path. With these extensions, we can manage the environment without the need for complicated awk and sed commands or other clumsy scripts. Here is an example of a simple modulefile for Design Compiler:

```
#%Module1.0
######################################################################
##
## dc_shell Modulefile
## by Mark Sprague
##
proc ModulesHelp { } {
        puts stderr "\tdc_shell - loads the environment for dc_shell\n"
}

module-whatis   "loads the environment for dc_shell"
setenv            SYNOPSYS          /tools/synopsys/dc/2000.11
prepend-path      LD_LIBRARY_PATH   $env(SYNOPSYS)/sparcOS5/dcm
prepend-path      PATH              $env(SYNOPSYS)/sparcOS5/syn/bin
set-alias         syn_help "acroread /tools/synopsys/sold/2000.11/top.pdf"
```

This example includes some basic module help information, sets the SYNOPSYS variable and adds the appropriate settings to PATH and LD_LIBRARY_PATH. It also shows an example of how to create a shell alias to invoke the online docs.

3.7 Creating Projects and Groups

Now that we have the infrastructure, we can complete the environment by adding projects and groups to the picture. Being able to access tools with a common interface is convenient but Modules is much more powerful than that. Suppose you have several projects going on simultaneously and you want to make sure that the correct version of every tool for every project is maintained. Furthermore, you would like the flexibility of adding new tools or new version of a tool without causing any ongoing work to be affected. Here's where the project and group modulefiles come in handy. Here is an example of a modulefile for a typical chip project:

```
#%Module1.0
######################################################################
##
## Sample project Modulefile
## by Erich Whitney
##
######################################################################

proc ModulesHelp { } {

        puts stderr "\foo - loads the project environment for project foo\n"
}
```

```
module-whatis   "loads the project environment for project foo"

#
# Put the path to your project's Verification Environment here
#
setenv VSRC_HOME /data/vsrc/foo
#
# Put the default version for each of your tools here:
#
setenv VCS_DEFAULT      6.0
setenv DC_DEFAULT       2000.11
setenv PT_DEFAULT       2000.11
```

By default, when a modulefile is loaded, the module search path is traversed and the first modulefile with the given name is used. If a hierarchical tool/version name is given (such as dc_shell/2000.11) then that specific version must be matched. However, if only the first level tool-name is given and there are multiple versions (like dc_shell/1999.10, dc_shell/2000.05 etc…) then Modules will pick the first one alphabetically. This may or may not be what you want depending on how you set up your modulefile naming convention. There *is* an explicit way to tell Modules which version you want to be the default. This is done with a .version file which is placed in the directory along with the modulefile. Here is a simple .vedrsion file you might put in your dc_shell modulefile directory:

```
#%Module1.0

set   ModulesVersion "2000.11"
```

Now any time someone issues the command:

```
% module load dc_shell
```

They will always get version 2000.11. However, this may not be what you really want. Suppose project X is using version 2000.05 and project Y is using version 2000.11. One will have to explicitly call out their version while the other could rely on the default. This is a dangerous situation that should be avoided. A simple solution to this problem is found by making a minor change to the .version file as shown below:

```
#%Module1.0

if [info exists env(DC_DEFAULT)] {
  set   ModulesVersion $env(DC_DEFAULT)
} else {
  set   ModulesVersion "2000.11"
}
```

Now if you look back at the project modulefile shown above, you'll see the variable DC_DEFAULT set to "2000.11". Each project can explicitly set which version of DC that is required and nobody has to remember to type the version number on the command line. Project X will get 2000.05 and project Y will get 2000.11 (assuming they set their DC_DEFAULT

variables appropriately). The advantage to method this is that it can be centrally maintained and put under RCS control.

What happens if a tool needs to be changed? First you can edit the modulefile to give an informational message notifying the users of an upcoming change and they will get this message the next time they load that modulefile. Modules also has some logging facilities so you can track a tool's usage and this information can be used to determine if and when an older tool may be removed or archived from the system. If a tool is removed, however, the modulefile can be modified to give an informational message telling the user that they'll have to go back to the archives, which is certainly more pleasant than, "Sorry, command not found!"

When a user sits down to do their work, they typically have to use several tools. From a Modules perspective, each tool should be maintained in a separate directory with its own modulefile thus making maintenance easier, but from the user's perspective, all they want to do is open up a shell and start working. This is where the group modulefile comes in handy. Here's an example group modulefile for doing synthesis:

```
#%Module1.0
######################################################################
##
## Synthesis Environment Modulefile
## by Erich Whitney
##
######################################################################

proc ModulesHelp { } {

        puts stderr "\tsyn - loads the synthesis environment\n"
}

module-whatis   "loads the synthesis environment"

module load dc_shell primetime
```

This file in conjunction with the project and tool modulefiles described above, makes the job of starting up the environment this easy:

```
% module load my_project syn
```

Now you can start making gates!

4 Multi-platform Support

The next step to building the environment is adding multi-platform support. Before going in to those details, here's a summary of how the environment works from user login to the shell prompt.

4.1 Starting a shell

In the environment we have presented, the user only has one initialization file. This is either .cshrc, .bashrc, or .profile, depending on their choice of login shell. In that file, the Modules init

script is called which sets up Modules. At this point, the control passes back to the user's initialization file. The environment is then examined to determine if this is a login from scratch or just a new shell starting up. If this is a fresh login, the default modulefiles, the user's personal modulefiles are optionally loaded and initialization is complete. If this is a shell that has been invoked from a parent shell, then the environment is inherited from that parent shell.

This is the point where multi-platform support needs to considered. If a child shell inherits all of its environment variables from the parent, then the possibility exists that some of those variables are platform-specific. If the users starts up a shell on a different platform from the parent shell, as is the case when using LSF, then it is likely that the environment will not be correct for the child shell.

With Modules, this problem is easily solved by performing a simple check in the user's initialization file for the existence of the "LOADEDMODULES" environment variable. This variable contains a list of all the modulefiles that are currently loaded in the user's environment. If this variable is defined during shell initialization, then all we have to do to update the environment is issue the "module update" command instead of "module load" and Modules will go through the list and reload the correct environment for the platform where it is being executed.

4.2 How does Modules work on multiple platforms?

At this point you might be inclined to think that the entire problem is solved and you can go on your merry way. But wait! There is one key piece of information missing from the story thus far. How does Modules know which platform it is running on and what to do about? This is where things start to get interesting.

Solving this problem is a bit like the old "chicken and the egg" riddle. You need to know what platform you are on before you load the modulefiles, but you need something in your modulefile to help you determine the platform. From the Modules perspective this problem is solved by the fact that Modules itself is started up by a shell script and you've had to build Modules for each platform on your network. Now you could modify the Modules startup scripts to set some variables and that tell you what platform Modules is running on but there is another way.

Our solution to this problem is to create a single modulefile that everyone in our environment must use. The obvious choice is the "modules" modulefile itself since everyone uses that modulefile anyway. There is a version of the "modules" modulefile that is built for you by the installation process. This file can easily be edited to add the commands necessary for multi-platform support. If you examine the man page for Modules, *modulefiles(1)*, you will see the Module command "uname". This version of "uname" is a platform-independent command that Modules has implemented to assist in multiple platform support.

The "uname" command has options that return the machine type, machine name, operating system revision, etc. You need to experiment a bit with this command to determine the options required to properly distinguish the platforms in your environment. Here is an example of the lines you would add to your "modules" modulefile:

```
setenv MODULES_MACH [uname machine]
setenv MODULES_OS   [uname sysname]
setenv MODULES_REL  [uname release]
```

On a Sun Sparc-10 running Solaris 7 you will get:

```
% printenv | grep MODULES_
MODULES_MACH=sun4u
MODULES_OS=SunOS
MODULES_REL=5.7
```

4.3 Creating multi-platform modulefiles

The next thing to do is make sure that everyone gets these variables. You can insert “module load modules” in the user’s initialization script. It is also a good idea to add a “prereq modules” line to each of your modulefiles that uses these variables. This will make your environment more robust and help to debug problems.

With these variables set in your environment, the typical modulefile changes slightly in that it has to determine which platform it is running on before setting the path. Here is a simple example showing how to improve our dc_shell modulefile to support Solaris and HP:

```
#%Module1.0
######################################################################
##
## dc_shell Modulefile
## by Mark Sprague
##
proc ModulesHelp { } {
        puts stderr "\tdc_shell - loads the environment for dc_shell\n"
}

module-whatis   "loads the environment for dc_shell"

setenv SYNOPSYS /tools/synopsys/dc/2000.11
switch –exact $env(MODULES_OS) {
  SunOS {
    set plat “sparcOS5”
    }
  HPUX {
    set plat “hpux10”
    }
  default {
    puts stderr “Error: Platform for OS $env(MACHINE_OS) not supported”
    exit
    }
}
prepend-path    LD_LIBRARY_PATH     $env(SYNOPSYS)/$plat/dcm
prepend-path    PATH                $env(SYNOPSYS)/$plat/syn/bin

set-alias syn_help "acroread /tools/synopsys/sold/2000.11/top.pdf"
```

NOTE: If you are using the HP platform, check the return value of “uname sysname” for the correct string to use in the switch statement shown above.

4.4 Multi-platform support for VCS and ACS

As it turns out, VCS supports multiple platforms already. The only issue you need to be aware of is building the correct "simv" executable for each platform. Most verification environments use some type of Make facility to build the simulation executables. If you use Modules to set up your path correctly, as shown above, then you will be able to support simulations on all of your machines.

On the other hand, ACS is a little bit more tricky. If you read the ACS User Guide you will find the "acs_dc_exec" and "acs_bs_exec" variables described in the section on customization. On the surface it would appear that these variables would allow you to properly choose which executable to call from ACS to run Design Compiler and Budget Shell. Unfortunately, ACS expects these variables to point to an executable during the preparation phase. This means that if you do an "acs_compile_design –prepare_only" on a Sun workstation and then try to submit the compile jobs to an HP machine, the Makefile will attempt to execute a Sparc binary on an HP machine with undesirable results. Using Modules, provides a trivial solution to this problem!

If we set up our dc_shell modulefile as shown above, then all we need to do to get the correct version of Design Compiler for whatever platform we are running on is:

```
% module load dc_shell
% dc_shell…
```

And if we're using LSF to run Design Compiler for us, all we need to do is:

```
% module load dc_shell
% bsub dc_shell…
```

What do we need to do to get ACS to work in our Modules environment? First look at how ACS calls Design Compiler. ACS looks at the "acs_dc_exec" and "acs_bs_exec" variables at runtime and determines if whatever the variable is set to is executable. ACS wants to see an explicit path to the Design Compiler executable. But in our Modules environment we just need to call "dc_shell" and the environment will do the rest. The answer is a simple wrapper script simply calls "dc_shell" with the same arguments and doesn't require an explicit path. Here is an example:

Create the following two scripts (don't forget to chmod +x the files):

/tools/scripts/dc_wrapper.sh:

```
#!/bin/sh
dc_shell $*
```

/tools/scripts/bs_wrapper.sh:

```
#!/bin/sh
budget_shell $*
```

And add the following lines to your .synopsys_dc.setup file:

```
set acs_dc_exec /tools/scripts/dc_wrapper.sh
set acs_bs_exec /tools/scripts/bs_wrapper.sh
```

Now ACS will happily call your wrapper scripts

5 What about license management?

Strictly speaking, Modules doesn't do anything for you with licenses. But licensing is an important part of building a complete environment. We solve the licensing problem with LSF since licenses are shared resources like CPUs, disks, memory, etc.

LSF doesn't manage your EDA licenses straight out of the box. We have configured our LSF installation as follows:

1. Create an ELIM file that creates a resource for each license we need to track. This script tells LSF how to check the license manager for the availability of a license and tracks the license usage within an LSF internal variable.
2. Create a set of queues for users that associates each tool (and license) with a class of machines capable of running that tool.
3. Users submit jobs into the queues for that tool which allows LSF to manage the license resources for all users.

This system works well as long as everyone uses it. If users insist on bringing up tools and grabbing licenses without using LSF, then it is possible that jobs in the LSF queue will not get the licenses they need when they run. This can be combated by adding LSF job-starter scripts, and other such techniques that are outside the scope of this paper.

6 Conclusions and Recommendations

As has been shown, using Modules to build your environment will solve most of the common problems with building and maintaining a design environment. Of course, modulefiles can be written that don't solve the problems.

In order to allow Modules to work efficiently, there are a few guidelines that need to be followed:

- Implement a well-defined directory structure.
- Create modulefiles for every tool in your environment.
- Construct your modulefiles to be platform-aware

Another aspect of creating a design environment is to encourage the use a distributed resource management (DRM) tool, like LSF, to manage all of your compute resources. Once users are accustomed to pushing all jobs through a DRM, there will be fewer license related issues.

Finally, why is this dragging your environment into the 90's? Modules was developed at Sun, and was originally presented in 1991. However, it's visibility into the hardware design community has been limited, even though it solves many of the environment issues that are

frequently encountered. Some CAD tool providers, including Synopsys, use Modules internally, but don't yet provide external support to automatically generate modulefiles when a release is installed.

7 Acknowledgements

We would like to thank John Mincarelli from Synopsys for his contribution to this paper. It was through John that we ultimately found out about Modules and his work at Synopsys was one of the inspirations for writing this paper. We would also like to thank Bob Gill of Axiowave for his help in setting up Modules on the system and for beating our Tcl installation into submission. Additional thanks goes out to John Furlani for his help in obtaining the original papers on Modules.

8 References

[1] John L. Furlani, "Modules: Providing a Flexible User Environment", *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V)*, pp. 141-152, San Diego, CA, September 30 – October 3, 1991.

[2] John L. Furlani, Peter W. Osel, "Abstract Yourself With Modules", *Proceedings of the Tenth USENIX System Administration Conference (LISA X)*, pp. 192-201, Chicago, IL, September 29 – October 4, 1996.

[3] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley Publishing Company, Inc., ISBN 0-201-63337-X, 1994.

Appendix A: Source file locations

The Modules package can be found at ftp://modules.sourceforge.net/pub/modules.

TCL or TCLPro can be found at www.scriptics.com.

For information regarding LSF, contact Platform Computing (www.platform.com).

Appendix B: .profile file

```
#
if [ "${USER:-}" = "" ]; then
  export USER=`/usr/ucb/whoami`
fi
loginshell=`ypcat passwd | grep $USER | awk -F: '{ print $NF }'`
loginshell=`basename $loginshell`
export HOSTNAME=`hostname`
export DEFAULT_MODULES="modules common perl lsf"
#
# Set mver to the desired Modules version
#
mver=3.1.2
case "$loginshell" in
bash)
  minit=/usr/local/Modules/$mver/init/$loginshell;;
[kz]sh)
```

```
  minit=/usr/local/Modules/$mver/init/$loginshell;;
*)
  minit=/usr/local/Modules/$mver/init/sh;;
esac

#
# This code should only execute if you're starting from scratch
# Otherwise, reload the environment
#
if [ -f $minit ]; then
  . $minit
  if [ "${LOADEDMODULES:-}" = "" ]; then
    module use /usr/local/Modules/modulefiles/groups
    module use /usr/local/Modules/modulefiles/projects
    module use /usr/local/Modules/modulefiles/tools
    module unuse /usr/local/Modules/modulefiles
    module load ${DEFAULT_MODULES}
  else
    module update
  fi
else
  echo "Modules not supported on this system"
fi

# If the user has additional customizations

case “$loginshell” in
bash)
  if [ -f ${HOME}/.bashrc_custom ]; then
     . ${HOME}/.bashrc_custom
  fi;;
*)
  if [ -f ${HOME}/.profile_custom ]; then
     . ${HOME}/.profile_custom
  fi;;
esac
```

Appendix C: .bashrc file

```
#
export HOSTNAME=`hostname`
export DEFAULT_MODULES="modules common perl lsf"
#
# Set mver to the desired Modules version
#
mver=3.1.2
minit=/usr/local/Modules/$mver/init/bash

if [ “${USER:-}” = “” ]; then
  export USER=`/usr/ucb/whoami`
fi
#
# This code should only execute if you're starting from scratch
# Otherwise, reload the environment
#
if [ -f $minit ]; then
  . $minit
  if [ "${LOADEDMODULES:-}" = "" ]; then
    module use /usr/local/Modules/modulefiles/groups
    module use /usr/local/Modules/modulefiles/projects
    module use /usr/local/Modules/modulefiles/tools
```

```
    module unuse /usr/local/Modules/modulefiles
    module load ${DEFAULT_MODULES}
  else
    module update
  fi
else
  echo "Modules not supported on this system"
fi

# If the user has additional customizations

if [ -f ${HOME}/.bashrc_custom ]; then
   . ${HOME}/.bashrc_custom
fi
```

Appendix D: Installation Notes

When we started this paper, we used Modules version 3.1.0. We subsequently installed and tested version 3.1.2. The 3.1.0 release has some bugs related to “module unload” and hierarchical modulefile names as well as at least one fatal that we have seen. The 3.1.2 release appears to address these issues.

Here are some notes on what we had to do to get Modules to build in our environment. You may or may not have to do these steps depending on how your environment was setup.

1. We needed to install TclPro1.4 from dev.scriptics.com in order to get this to build properly. TclPro1.4 includes Tcl/Tk 8.3 that we needed to build Modules. It is installed by downloading the tar.gz from scriptics.com, extracting it and running the resulting setup.sh file. We installed the software under /usr/local/tclpro. Note that part of TclPro1.4 is a licensed product which means you should read the license agreement. It may be possible to configure Tck/Tk8.3 another way.
2. It was necessary to edit RKOConfigure in the source directory, set TCLTKROOT to /usr/local/tclpro and change --with-tcl-libraries to be $TCLTKROOT/solaris-sparc/lib (it is normally $TCLTKROOT/lib). Also, you may need to edit the location of the X11 directory. On Linux, for example, it needs to point to /usr/local/X11R6. If you read the INSTALL file in the Modules distribution it recommends running RKOConfigure and I would recommend it as well since there are too many options to ‘configure’ to remember.
3. Next run RKOConfigure (./RKOConfigure). It produces the Makefile.
4. Edit the Makefile, search for LIBS line and change "-tcl " to "-tcl8.3" (again no quotes). This step wasn’t necessary for the Linux build.
5. Run "make".
6. Run "make install".