

Tcl: The Good, The Bad, and The Ugly

Gregg D. Lahti

gregg.d.lahti@intel.com

Steve J. Brown

steve.j.brown@intel.com

Intel Corporation

ABSTRACT

Tcl is rapidly becoming the EDA industry's choice of language to facilitate a versatile user interface. Design Compiler and Primetime now support Tcl as the primary user interface, and many engineers are starting to convert to Tcl-based synthesis and static timing.

This paper will show real-world examples, advantages, and pitfalls of using Tcl to drive Synopsys tools, based upon the experience from creating a Tcl Oriented Procedural Synthesis (TOPS) environment. The examples are a working tutorial on how Tcl was used to automate various operations and the basic user errors, misconceptions, and problems that were encountered.

1.0 Overview

Tcl is short for the Tool Control Language founded by John Ousterhout in 1987. Tcl satisfies the need for a common, simple, and easy-to-implement language parser for applications. Many applications in the EDA world are adopting Tcl as the de-facto standard for a programmable user interface due to the ease of integration of the parser into the tool and the ease-of-programming of the language. There have been many revisions of the language up to today, but the core essentials haven't changed much.

Implementing Tcl into the Synopsys family finally allows a real programming interface into a highly complex tool and reduces the need to shell out to standard UNIX utilities such as sed, Perl, and grep, to handle items within Design Compiler. Using Tcl enables more complex operations and methods of handling data than the standard DCSH could utilize.

This paper uses the 2000.05 release of Design Compiler for the examples. It is our hopes that we can show first time and experienced users of Design Compiler how Tcl is used through examples and some of the main problems of writing Tcl code and translation from DCSH to Tcl-based scripts can be avoided.

2.0 The Good: Tcl Makes Synthesis Easier

2.1 Naming Convention

Regardless of the language it is necessary to standardize on a naming convention for those particular languages constructs. Whether they are signals, ports, variables, procedures, or functions developing a clear, concise, and readable convention can save hours of debugging or providing support for unreadable code. In our Tcl-based synthesis environment we adopted the convention of using P_* for procedure names and G_* (all caps) for global variables. Variables local to procedures were all lower case. A nicety of Tcl is that one can use the `info` command to report information about the internals of the Tcl interpreter, such as listing all the global variables or user defined procedures.

<pre>proc P_addlist {element} { global G_LIST lappend G_LIST \$element } proc P_viewlist {mylist} { upvar 1 \$mylist G_LIST puts "\$mylist = \$G_LIST" }</pre>	<pre>%info proc P_* P_viewlist P_addlist %info vars G_* G_LIST %</pre>
---	--

Defined Procedures

info Command Results

The above example shows how to list all the defined procedures and global variables adhering to the naming convention using the `info` command. The Synopsys `help` command can also be used to provide information about the procedures in much the same way as `info` if the `-verbose` switch is included. A more detailed description of command parameters and usage is displayed providing the `define_proc_attributes` command has been used as explained in section 2.3.

2.2 Procedures

Tcl procedures were the basis for our Tcl synthesis environment, they allow the grouping of one or more commands allowing a natural modularization of code. Within Tcl all variables set outside a procedure are global, while variables set within procedures are local to the procedure and only exist during the execution of the procedure. Variables can be made available to commands outside the procedure by using the `global` or `upvar` commands. Tcl allows the passing of variables to procedures by reference using the `upvar` command. Procedures allow the setting of default and variable number of arguments, which is demonstrated within the next example.

<pre>proc Pass {{arg1 foo} {arg2 bar} args} { puts "arg1=\$arg1, arg2=\$arg2, args=\$args" }</pre>	<pre>%Pass arg1=foo, arg2=bar, args= %Pass the arg1=the, arg2=bar, args= %Pass the good bad ugly arg1= the, arg2= good, args= bad ugly</pre>
Procedure	Using a Procedure

Making global variables visible within a procedure is accomplished by using the `global` command. The `upvar` command in this example is used to pass the variable `list` to the `viewlist` procedure by name instead of by value.

<pre>proc Addlist {element} { global list lappend list \$element } proc Viewlist {mylist} { upvar 1 \$mylist list puts "\$mylist = \$list" }</pre>	<pre>%Addlist gunfighter gunfighter %Addlist outlaw gunfighter outlaw %Viewlist list list = gunfighter outlaw</pre>
Procedure	Procedure Results

2.3 Documenting User Procedures

Synopsys uses attributes on procedures to provide help or information about the procedure. This feature can be exploited further in user-defined procedures, allowing help information to be displayed using the `help P_proc` command in Design Compiler. To setup the attribute, the

`define_proc_attribute` command is used with the appropriate help string and any defined arguments that are required for the procedure. The following example is the info line for the `P_get_license` procedure set by the `define_proc_attribute` that accepts only one argument for the license name.

<pre>define_proc_attributes P_get_license \ -info "TOPS: Procedure to get a license" \ -define_args { {ln "license name" ln string required} }</pre>	<pre>dc_shell-t> help -v P_get_license P_get_license #TOPS: Procedure to get a license ln (license name) dc_shell-t></pre>
Procedure	Command Results

2.4 Variables

Tcl variables are evaluated as strings in the language. This is very different than other programming languages such as C or Perl. Since variables are evaluated as strings within Tcl, so the concept of types (integer, Boolean, etc) is not available in the Tcl language, but Boolean operations can be performed on strings. Variables are always local in scope to the procedure or main script.

(Referencing variables in DCSH is automatic, there is no \$ for evaluation. If the variable was declared (e.g. `foo = junk`) and then referenced (e.g. `goo = foo`) `goo` would be set to “junk”. If a variable was set to a non-variable, then it is treated as a string and no error occurs (e.g. `blech = barf`) then the variable gets assigned the string occurs (e.g. `blech = barf`) then the variable gets assigned the string “barf”.

In DCSH, variables that weren’t declared could be referenced without errors. In Tcl, referencing a variable that hasn’t been declared will result in an error. This is somewhat annoying and will usually crash any converted DCSH scripts using the Synopsys transcript program if the variable has not been declared in the file or procedure that it is being used in. One approach to get around this problem is to set the variable with a null value before using the variable.

<pre>## fist_full_of_dollars isn't declared set outlaw \$fist_full_of_dollars</pre>	<pre>set fist_full_of_dollars "" ## fist_full_of_dollars declared, no error set outlaw \$fist_full_of_dollars</pre>
Fails, fist_full_of_dollars is not declared	A method of setting NULL variables

In the above example on the left, the variable `fist_full_of_dollars` was not set and caused an error. The example shown on the right what will work because the variable gets declared with nothing (a null value) before it is evaluated in the following `set` command.

However, this method could over write the variable being referenced in another script or procedure depending upon if the variable was declared global elsewhere. A safer alternative

method of determining if a variable is set is to use the `info` command. The `info` command is a pretty useful item that has a list of modifiers to further define the action of the `info` command. An example to determine if a variable has been declared is shown in the following example.

```

if {[info exists for_a_few_dollars_more]} {
    expr $for_a_few_dollars_more + 1
} else {
    set for_a_few_dollars_more 1
}

```

Using the info Command

Either method as described above will work to avoid undeclared variables.

Programming tip: when referring a variable in Tcl the potential to error the parser exists if the variable is not declared. When in doubt, use the `[info exists variable]` command to verify if the variable is declared. This is extremely useful when converting DCSH scripts to Tcl with the transcript program.

2.5 Hashes and Associative Arrays

Like Perl, Tcl incorporates the use of associative arrays that use a string as the index or key to the array elements. Unlike lists, which are stored as a linked list in memory, arrays are stored internally as hash tables, which allow each element of the array to be accessed with relatively the same cost. This is especially important when dealing with large data sets where access times may be very large. Arrays are particularly useful when dealing with complex data structures or for grouping together a set of related variables. The following examples demonstrate the use of arrays.

<pre> set myarray(actor) eastwood set myarray(tv) rawhide </pre>	<pre> %set myarray(actor) eastwood %set myarray(tv) rawhide % </pre>
--	--

Commands

Command Results

More complex data structures such as records can be created, but before we go much further it is necessary to introduce the `array` command which returns information about the array. The `array exists` command returns 1 if the array exists and 0 otherwise.

<pre> array exists myarray </pre>	<pre> %array exists myarray 1 % </pre>
-----------------------------------	--

Commands

Command Results

Other array commands include `get`, `names`, `set`, `size`, `startsearch`, `nextelement`, `anymore`, and `donesearch` of which only `get` and `names` will be discussed here. The array `get` command returns an alternating whitespace separated list of array keys and data (e.g. `key data key data ...`), and the array `names` command returns a whitespace separated list of keys within the array.

<pre>array get myarray array names myarray</pre>	<pre>%array get myarray actor eastwood tv rawhide %array names myarray actor tv</pre>
Commands	Command Results

The following example shows how a procedure can be used to encapsulate a data structure and hide the implementation of the data structure from the user.

<pre>proc P_AddMovie {name star genre} { global movieGenre movieStar set movieStar(\$name) \$star set movieGenre(\$name) \$genre } proc P_ListMovieInfo {movies} { global movieGenre movieStar foreach name \$movies { puts "Title:\$name" puts "Star :\$movieStar(\$name)" puts "Genre:\$movieGenre(\$name)\n" } }</pre>	<pre>%P_AddMovie {Hang em High} {Eastwood, Clint} Western Western %P_AddMovie Unforgiven {Eastwood, Clint} Western Western %P_ListMovieInfo {Unforgiven {Hang em High}} Title :Hang em High Star :Eastwood, Clint Genre :Western Title :Unforgiven Star :Eastwood, Clint Genre :Western</pre>
Declared Procedures	Command Results

2.6 Regular Expressions With `regsub` and `regexp`

Regular expressions did not exist in DCSH . To do any regular expression operations in the Design Compiler, the user had to exec out to a shell with the `info` and run `Perl`, `awk`, `grep`, or `sed` to do the regular expression work. In `Tcl`, the regular expression syntax is pretty close to UNIX-style `sed` or `awk`, so the characters used in regular expressions match some of the `Tcl` language constructs. This causes some confusion in the parsing of the code when using regular expressions with the `[]` and `$` characters. When using these characters in regular expressions, use quotations or separators such as `""` or `{ }` to group the regular expression. Remember that grouping with double quotes allows substitutions while grouping with curly braces prevents substitutions.

For example, let's use a piece of the `P_get_license` procedure listed in section 5.2 that gets the output of the `list_license` into a variable so we can process on it. In the `P_get_license` procedure, the `$licenses` contains the output of the `list_license` command. However, the output from the command is multi-line and contains some extra text:

<pre>dc_shell-t> list_license Licenses in use: Design-Compiler VHDL-Compiler dc_shell-t></pre>	<pre>dc_shell-t> set licenses [list_license] Design-Compiler VHDL-Compiler dc_shell-t> set \$licenses Design-Compiler VHDL-Compiler</pre>
--	--

list_license Command

What list_license Really Should Do

What the `list_license` command really should do is return a list of licenses in Tcl list form. Unfortunately, this isn't the case: the `list_license` command isn't variable friendly and only prints out the licenses used to stdout rather than to a variable. To fix this limitation our script must redirect the output of the `list_license` command to a file and then cat the file contents into a variable.

```
# now that we have a TCL variable with the list_license output,
# strip out the crud that DC uselessly puts in and parse the list
regsub {Licenses in use:} $licenses {} licenses;
foreach checkedout_license $licenses {
    if [string match $checkedout_license $ln] {
        set fetch_license 0;
    }
}
```

Section of P_get_license Procedure

However, the output now contained in the variable `$licenses` is still multi-line and needs massaging in order to use the information. We use the `regsub` command in Tcl to strip out the useless header info. Then we operate on the variable using a `foreach`, since the `foreach` command uses whitespace as a delimiter between items (remember that tabs, spaces, and carriage returns are defined as whitespace characters). The `string match` command is used to see if we have already checked out the license we're trying to check out. Why do this extra command? This extra code is required because the `get_license` command is broken and will incorrectly provide an error status if we try and check out a license that we already have. A user shouldn't care about re-checking out a license. However, a user will care if they **do not** get the license. In this example, the `regsub` and `string match` commands have enabled a grammatical workaround to a Synopsys "feature".

An example of using the `regexp` command is our `P_proj_insert_buffers` script located in section 5.4. The goal of this procedure is to find hold violations reported by the `report_constraint` command and add in a user-defined cell, such as a buffer, to the end of the path before the violating cell to add hold time. See section 5.4 for a complete listing of the code.

In the middle of the code, we use the `regexp` command to get the instance name of the violator cell and to get the path hierarchy to the cell.

```

# Fix all the violations for the violator list
foreach reg_cell $all_violator_list {

    # Get the instance name of the violator
    regexp {(.*)/.*} $reg_cell var1 var2
    set inst_name [file tail $var2];

    # Get the hierarchy path to the violating register
    regexp {(.*)/.*} $var2 var1 hier_path
    ...

```

A regexp Example

Note that braces are used to contain the regular expression we wish to operate with.

Programming tip: if you are using special escaped characters such as `\t` or `\n`, put double-quotes “” around the string so the Tcl parser will evaluate the string correctly.

2.7 Time Functions and File I/O

We’ve found it very useful to timestamp specific operations during synthesis to profile time spent in various functions and aid in determining synthesis bottlenecks. To handle this operation, we use a procedure to print out specific information of the hostname, username, and the current wall-clock time to the DC console that gets logged to a file.

```

proc P_timestamp {} {
    set c [clock format [clock seconds]];
    set h [sh hostname];
    set u [sh whoami];
    echo "#TIMESTAMP: " $c " " $h " " $u;
}; # end P_timestamp;

```

Timestamp Procedure

The clock command has multiple arguments for formatting the time output and the time units that can be used in a variety of ways. The `P_timestamp` procedure uses two nested clock commands to print the current time in a specific format.

File I/O is much more standardized in the Tcl language than it was in DCSH. Since Design Compiler lives in a multi-OS environment (UNIX, Windows NT, Linux?), the `file` operation commands in Tcl allow basic file functionality in a portable environment without knowing the which OS it is running on. For example, the directory separator for Windows is `\` character, for UNIX it is the `/` character. Your script can be OS-tolerant if you use the built-in file operations.

```
set my_library libs/$my_lib
```

Bad Use of Path Declaration

```
set my_library [file join libs $my_lib]
```

Using file join for Path Declaration

In the previous example, using the file join command will figure out which directory separator to use by the OS. The left example shows a hard-coded form that will break on a Windows platform.

There are many file operators that are supported in Tcl that one had to shell out of DCSH with a UNIX command to emulate. One of the operators that we use in the `P_proj_insert_buffers` is the `file tail` command to get at the trailing component of a pathname.

Programming tip: use the `file arg` commands when manipulating files or file attributes. They work well and are portable across all OS variants.

Accessing files in Tcl is similar to C in that it requires that you open and close the file being accessed. The following is an example of our `P_log_filter` which extracts all warnings, errors and elapsed time messages from the log file and prints them to a summary file. This example demonstrates the basic Tcl `open`, `gets`, `puts`, `and`, and `close` commands.

In the example below the variable declarations we see that the `open` command is used to set variables for the file handles of the files being read and written. The first file is being opened with the read only attribute “r”, and, and the second with the write only attribute “w”. Within the while loop we see the use of `gets` with two arguments, the first is the file handle, and the second is the variable which will contain the returned line minus the newline. After processing all of the lines within the log file we see the use of the `puts` command which, like the `gets` command, has two arguments: the file handle and the string of text to print. The `puts` command will append a newline character to the string being written. The first argument to the `puts` command is left out which tells the `puts` command to default to stdout.

```
#####
# usage: P_log_filter {log_path unitname}
#####
proc P_log_filter {log_path design} {
  set log_file $log_path/${design}.log;
  set filter_log_file $log_path/${design}.log.summary;
  set error_cnt 0; #number of error messages
  set warning_cnt 0; #number of warning messages
  set line_cnt 0; #number of line in log file
  set elapsed_cnt 0; #number of elapsed messages

  set LOG_HANDLE [open $log_file r];
  set FILT_LOG_HANDLE [open $filter_log_file w];

  # Search for lines with errors and warnings with the write only attribute
  while {[gets $LOG_HANDLE line] >= 0} {
    incr line_cnt 1;
    if {[regexp {^Error} $line]} {
      regsub Error $line " " new_line;
      set errors($error_cnt) "line no ${line_cnt}${new_line}";
      incr error_cnt 1;
    }
  }
}
```

```

    } elseif { [regexp {^Warning} $line]} {
        regsub Warning $line "" new_line;
        set warnings($warning_cnt) "line no ${line_cnt}${new_line}";
        incr warning_cnt 1;
    } elseif { [regexp {Elapsed} $line]} {
        set elapsed($elapsed_cnt) $line;
        incr elapsed_cnt 1;
    }
}

# Print Errors and Lines in a filtered log
puts $FILT_LOG_HANDLE "There are $error_cnt errors, $warning_cnt warnings";

# Print the Elapsed messages if any
if {$elapsed_cnt > 0} {
    puts $FILT_LOG_HANDLE "\nElapsed :";
    for {set i 0} {$i < $elapsed_cnt} {incr i} {
        puts $FILT_LOG_HANDLE $elapsed($i)
    }
}

# Print the errors if any
if {$error_cnt > 0} {
    puts $FILT_LOG_HANDLE "\nErrors :";
    for {set i 0} {$i < $error_cnt} {incr i} {
        puts $FILT_LOG_HANDLE $errors($i);
    }
}

# Print the warnings if any
if {$warning_cnt > 0} {
    puts $FILT_LOG_HANDLE "\nWarnings :";
    for {set i 0} {$i < $warning_cnt} {incr i} {
        puts $FILT_LOG_HANDLE $warnings($i);
    }
}

close $FILT_LOG_HANDLE;
close $LOG_HANDLE;
}

```

The P_log_filter Procedure

2.8 Sockets (How to Fix the Lack of Tk)

Sockets in Tcl are network-like communication channels based upon TCP. A socket is accessible like a pipe or through an open command, much like a file operation. Sockets become very useful if the need arises to communicate between computing machines across a network or to communicate between programs. Programming sockets relies on the client-server model: a server socket runs in the background and waits for a client program to start and connect. Communication between the server and client can be bi-directional.

One good use of sockets is to overcome the missing Tk toolkit for building graphical images. The Tk toolkit allows a user to easily create custom graphical user interfaces (GUIs). Synopsys did not integrate the other half of the Tcl language, Tk, into Design Compiler. However, you can overcome this huge shortcoming in Design Compiler by writing Tcl scripts that utilize sockets for communication. The method to do this would be to use a server-side Tcl script that utilizes the built-in Tk toolkit for GUI's

and implements a socket for communication. A Design Compiler Tcl script would then communicate to the server-side Tcl script through the socket connection, passing information back and forth to the server-side script.

For example, the following two Tcl scripts could be used to create a window that displays information sent from the Design Compiler session. The `server.tcl` script builds a frame-per-socket connection request in the window, so multiple frames of information can be created in the window by establishing multiple socket connections by the client. The `server.tcl` example is using a hardcoded port of 9996 to communicate through. Remember that any port over 1024 is usable by the user without special privileges.

```
#!/usr/local/bin/wish
#####
# $Id: server.tcl,v 1.1 2000/06/27 18:40:37 glahti Exp $
#####
# filename:      server.tcl
#  author:      Gregg D. Lahti
#  created:     06/26/00
#####
# description:  server socket program, builds a window that we can pass info to
#              from DC. New text frames can be added by creating new sockets
#              start this script as "wish -f server.tcl 9996"
#####

proc buildmain {} {
    wm title . "TOPS Logviewer"
    wm protocol . WM_DELETE_WINDOW { set closewindow 1 }
}

proc buildframe {sock} {
    global tframe
    set tframe [frame ".$sock"]           ;# Create a text widget to log the output
    set $tframe.log [text $tframe.log -width 80 -height 10 \
        -borderwidth 2 -relief raised -setgrid true \
        -yscrollcommand {$tframe.scroll set}]
    scrollbar $tframe.scroll -command {$tframe.log yview}
    pack $tframe.scroll -side right -fill y
    pack $tframe.log -side left -fill both -expand true
    pack $tframe -side top -fill both -expand true
}

proc display {sock line} {
    global tframe
    .$sock.log insert end $line\n
}

proc StartServer {port} {
    global echo
    set echo(main) [socket -server SocketAccept $port]
}

proc SocketAccept {sock addr port} {
    global echo
    set echo(addr,$sock) [list $addr $port]
    fconfigure $sock -buffering line
    fileevent $sock readable [list GetFromClient $sock]
    buildframe $sock
}
```

```

}

proc GetFromClient {sock} {
    global tframe echo log
    if {[eof $sock] || [catch {gets $sock line}]} { ;# end of file or wierd
drop
        close $sock
        unset echo(addr,$sock)
    } else {
        if {[string compare $line "closeit"] == 0} {
            # Prevent new connections, existing connections stay open.
            close $echo(main)
        } else {
            display $sock $line
        }
    }
}

}

#####
# Start of Main program
#####

set port [lindex $argv 0]
if { $port == "" } {
    puts stdout "Errorr:  invalid port!  \n";
    puts stdout "usage:  server.tcl [port]\n";
    exit 1;
}
buildmain            ;# create & label window
StartServer $port   ;# start up server & listen on socket
vwait closewindow   ;# handle events
exit 0

```

Server Socket Script server.tcl

```

#!/usr/local/bin/wish
#####
# $Id: client.tcl,v 1.2 2000/06/27 18:43:42 glahti Exp $
#####
# filename:      client.tcl
#  author:      Gregg D. Lahti
#  created:     06/27/00
#####
# description:  example client procedure and commands.  Include this in the
#              DC session or run standalone.
#####

proc Echo_Client {host port} {
    set s [socket $host $port]
    fconfigure $s -buffering line
    return $s
}

exec server.tcl 9996 &                ;# start server program
set host [info hostname]              ;# get host name
set socket [Echo_Client $host 9996]   ;# open socket
puts $socket "Hello!"                 ;# stuff some data to socket
# open another socket, get a new text frame
set socket2 [Echo_Client $host 9996]  ;# open socket2
puts $socket2 "Hello again!"          ;# stuff some data to socket2

```

Client Socket Script client.tcl

The `client.tcl` script executes the server script in the background. The server script sits and waits for a socket connection to be opened. The client opens the socket by executing the `Echo_Client` procedure and setting the returned value to a variable called `socket`. Using the `puts` command will stuff the string data to the socket. Once the socket is opened at the server end, a frame is built and information passed to the socket gets printed in the frame.

3.0 The Bad: Tcl Quirks That Trap Experienced Programmers

3.1 The Use of “”, {}, and []

Braces and quotes allow the separation of lists, characters, and other items within the Tcl language. These items allow the parser to group items within the braces or double quotes. The following examples are identical in setting a variable with a string content:

```
set movie {High Plains Drifter}
set bar "High Plains Drifter"
```

Example of Using Braces and Quotes

Using quotes allow variable substitution. Curly braces prevent substitutions. This applies to any command, variable, and backslash substitutions. The next example won't get variable expansion which will result in the variable `movie` being set to “\$bar”.

<pre>set movie {\$bar}</pre>	<pre>%set movie {\$bar} \$bar %</pre>
------------------------------	---------------------------------------

Command

Command Result

The next example, however, will expand the variable:

<pre>set outlaw {Jose Wales} set movie "The Outlaw \$outlaw"</pre>	<pre>%set outlaw {Jose Wales} Jose Wales %set movie "The Outlaw \$outlaw" The Outlaw Jose Wales %</pre>
--	---

Commands

Command Results

Brackets are very different from the quote and braces. Brackets allow execution of commands, rather than provide a grouping function. Commands inside the brackets get evaluated and executed, with any results being passed back into the calling line function. Note that the Tcl parser treats entire lines as a command group, with any nested brackets getting executed and evaluated first.

<pre>set mystring {Paint Your Wagon} set line_length [string length \$mystring]</pre>	<pre>%set mystring {Paint Your Wagon} Paint Your Wagon %set line_length [string length \$mystring] 16 %</pre>
Commands	Command Results

Programming tip: grouping occurs before substitution. The Tcl parser groups first, then executes or substitutes variables.

3.2 Global Variables

Contrary to conventional C/C++ programming, variables defined at the top level of a Tcl program are not accessible inside procedures without defining them again in the procedure as a global variable. This annoying feature can cause programmer frustration and countless hours of debugging.

To reference a global variable within a procedure, declare the variable as `global` inside the procedure. Declaring a global variable at the top level of the Tcl program does nothing.

<pre>proc setit { myvar } { global mystring set mystring \$myvar } proc printit {} { global mystring echo "mystring is: \$mystring" }</pre>	<pre>%setit {Pale Rider} Pale Rider %printit my string is: Pale Rider %</pre>
Commands	Command Results

In this example, the `setit` procedure sets the global variable to a string. The `printit` routine accesses the global variable and prints it to the screen.

Programming tip: global variables work differently when compared to C/C++ and should always be declared with the `global` command inside a procedure.

3.3 Strings and Lists

In Tcl, everything is evaluated as a string. For example, any mathematical operations require the `expr` command to do the actual math operation with the resulting value from the `expr` command formatted back into a string. Strings are the basic element in Tcl, and there is a multitude of commands to manipulate and evaluate strings. The `string` command has an array of operations that can be used to

manipulate strings. For comparing strings, the most reliable command to use is the `string compare` function.

3.4 Line Termination

Line terminations are sometimes necessary in helping the Tcl parser understand how to interpret the command. The line termination command is a semi-colon, “;”. An example of using line termination is shown in the following example.

<pre>DCSHell-t> set bar [expr 3 * 4] # test mult Error: wrong # args: should be "set varName ?newValue?" use error_info for more info. (CMD-013)</pre>	<pre>% set bar [expr 3 * 4] ;# test multiply 12</pre>
---	---

Causes Error in the Tcl Parser

Doesn't Break Tcl Parser

In this example, a line termination character is required to signal the Tcl parser that the following could be interpreted as another line or operation. Otherwise, the comment character and following string will be interpreted as another argument to the `set` command. This yet another annoyance in using the Tcl language due to the limited parser and the error message will be cryptic.

Programming tip: put a semi-colon before the comment character (i.e. `;`). This eliminates a lot of debugging headaches and doesn't hurt Tcl code execution.

3.5 Oddities of Conditionals and Braces

If you are used to a Kernighan and Ritchie programming style, then the `if/elseif/else` programming constructs in Tcl won't cause you too many headaches. When doing conditional programming in Tcl, the parser must see a specific brace structure for the else condition or it will complain and die with a cryptic warning.

In the left example, the beginning brace “{” in the `if` construct is moved to the second line. This will break the Tcl parser. The correct construct is shown on the right.

<pre>set death_valley 1 if {\$death_valley == 1} { echo "death_valley was set to 1\n" }</pre>	<pre>set death_valley 1 if {\$death_valley == 1} { echo "death_valley was set to 1\n" }</pre>
---	---

Incorrect Brace Position for If Construct

Correct Bracing for If Construct

The Tcl parser in Design Compiler will complain with a very cryptic warning:

```

Error: wrong # args: no script following "$death_valley == 1" argument
      use error_info for more info. (CMD-013)
Error unknown command `
      echo "death_valley was set to 1\n"
` (CMD-005)

```

Holy cow! The error message was long and not very informative to the fact that the beginning brace “{” is on the next line following the if statement instead of on the same line. At least Design Compiler provided an `error_info` command to dump out more info that can be useful for tracing out nasty syntax errors. However, even the `error_info` command won’t tell you exactly what went wrong on the previous example.

The same problem goes for the `elsif` and `else` conditionals. If the braces aren’t in the right spot, the Tcl parser will fail.

<pre> set death_valley 1 if {\$death_valley == 1} { echo "death_valley was set to 1\n" } else { echo "death_valley was set to 0\n" } </pre>	<pre> set death_valley 1 if {\$ death_valley == 1} { echo "death_valley was set to 1\n" } else { echo "death_valley was set to 0\n" } </pre>
---	--

Incorrect Brace Position for else Construct

Correct Bracing for else Construct

In the example on the left, the error message was less verbose and still vague:

```

Error: unknown command `else' (CMD-005)

```

Design Compiler didn’t even offer you an `error_info` message for this error. Just remember that the Tcl parser is somewhat limited due to the nature of how it works on lines rather than multiple lines and a command separating character like other languages. Where you put the braces matters in the Tcl parser.

Programming tip: put the beginning braces of the condition at the end of the line containing the conditional declaration, put any ending braces from the previous condition at the beginning of the next line.

4.0 The Ugly: Synopsys-specific Implementations of Tcl

There are items that are needed for synthesis and general functionality, but are handled in an odd method through the Tcl language. Here are some of the weird items of using Synopsys that need more explanation and examples than what the documentation contains.

4.1 Collections

Collections are not part of the Tcl core language. Collections are Synopsys-specific programming items that allow attributes of a Synopsys-specific item to be grouped into a single variable reference, using similar Object Oriented Programming concepts to C++ where different items can be grouped together in a class-like element. Synopsys commands generally return a list of collections rather than a string or list. This is important to know, since collection items need to be operated on differently than strings or lists.

Collections are referenced as a string handle and cannot be operated on like a list. Instead, Synopsys-specific commands such as `add_to_collection`, `foreach_in_collection`, and `remove_collection` allow operations on the collections. For example, if a design had three input ports, `clk`, `rst_n`, and `capture`, the `all_inputs` command returns a collection not a list. To access each item, collection commands must be used.

<pre>% set in_list [all_inputs] {"clk", "rst_n", "capture"} % set item0 [lindex \$in_list 0] {"clk", "rst_n", "capture"} % set foo {"clk", "rst_n", "capture"} %</pre>	<pre>% set in_list [all_inputs] {"clk", "rst_n", "capture"} % set item0 [index_collection \$in_list 0] {"clk"} % set item0 {"clk"} %set port_name [get_object_name \$item0] clk % set \$port_name clk</pre>
--	---

Incorrect Method of Accessing Collections

Accessing Collections Correctly

In the left example above, the `all_inputs` command returns a collection, not a list and cannot be accessed like a list. Notice that the variable `item0` is set with the collection results, rather than the first item in the collection (or the 0th element). In the right example, the `index_collection` is used to get at the 0th element of the collection that just happens to be the clock pin. Unfortunately, the `item0` variable is also a collection, so the extra step of assigning the `port_name` variable must incorporate the `get_object_name` command on the collection variable `item0`.

Programming tip: accessing collections requires the use of the special procedures or flag options. Collections are not Tcl lists!

4.2 Attributes

Another important item to remember about collections are that there are many Synopsys-defined attributes that get “attached” to the items in a collection. Attributes help define the item with special synthesis information that Design Compiler can utilize. Attributes are grouped into the following categories:

cell	clock	design
library	net	pin
port	read_only	reference

List of Synopsys Attributes

Some attributes are read-only and set/reset by Design Compiler for informational purposes. Some attributes may be set by the user using the `set_attribute` command. The user can also create his/her own attributes which can be helpful for tagging items with more information. In the following example, the `reset_name` variable is set with a user-defined attribute called `is_reset`.

```
set reset_name "sysrst";
echo "#RESET-Info: Setting reset attributes on port - $reset_name";
remove_driving_cell $reset_name;
set_ideal_net $reset_name;
set_drive 0 $reset_name;
set_false_path -from $reset_name;
set_attribute $reset_name is_reset true -type Boolean;
```

Using the `set_attribute` Command

This is a simplified example of setting up a reset port on a design with synthesis parameters and a user attribute of Boolean type. This attribute can then be used within the synthesis job to so we can handle the synthesis constraints differently for reset pins.

Programming tip: doing a “man attribute” in Design Compiler will list all Synopsys-defined attributes that can be utilized.

4.3 Accessing Collections and Attributes

Synopsys has provided access to the contents of collections through the use of `sort_collection`, `filter_collection`, and `foreach_in_collection` operations. Some commands can optionally pass a `-sort` or `-filter` command with an argument to further sort or filter on specific items. The following shows an example of finding a specific cell using the `filter_collection` command:

```
filter_collection [get_cells *] "ref_name == FD1S"      ;# find flops called FD1S
in design
```

Using the `filter_collection` Command

The `-filter` option in commands acts much like the `filter_collection` command and can be used in a variety of situations. For example, to find all cells in the design that are not mapped, the following command can be used.

```
set unmapped_cells [get_cells -filter {@is_unmapped == true} "*"];
```

Using the `-filter` Command

We can also reference the user-defined attribute, `is_reset`, using the `-filter` command as well:

```
set clk_ports [get_ports -filter {@is_clock == true} "*"];  
set rst_ports [get_ports -filter {@is_reset == true} "*"];
```

Filtering for User Attributes

The `foreach_in_collection` command is useful for looping through all of the collection items and operating on each item set within a collection. A useful procedure to count the number of cell instances using the `foreach_in_collection` command is shown below.

```
proc P_proj_cell_cnt {} {  
    set leaf_cell "";  
    set cell_list_ptr [filter [find -hier cell "*"] "@is_hierarchical == false \  
&& @ref_name != \\"**logic_0**\" && @ref_name != \\"**logic_1**\""];  
    foreach_in_collection entity $cell_list_ptr {  
        set cell_name [get_object_name $entity];  
        lappend leaf_cell $cell_name;  
    }  
    return [llength $leaf_cell];  
}; # end P_proj_cell_cnt
```

Counting Cells Example

In this example, the `cell_list_ptr` variable gets set to a collection of cells that aren't hierarchical or VSS/GND pins. Note that the wildcard character "*" is used to grab all of the cells in the `find -hier` command and the use of special operators in the `filter` command for determining reference names and if the value is a hierarchical reference name. The `foreach_in_collection` command operates on the collection and steps through each element in the collection and sets the variable `cell_name` with the cell name using the `get_object_name` command. The `leaf_cell` variable gets appended with the cell name, causing the variable to set like a list of items. Finally, the `llength` command counts the number of items in the list (really cell instances) in the `leaf_cell` variable and returns this number as the final cell count.

4.4 Collections Gotchas

In DCSH programming, it was easy to take a list of items and subtract elements or another list from the first list. For example, a list of all inputs without the clock inputs could be done with the following command:

```
in_list = all_inputs() - all_clocks()
```

DCSH to Get Just Input Signals Without Clocks

In the DCSH example above, getting just the input signals of a design without the clocks was easy to do by issuing the `all_inputs()` command and subtract the `all_clocks()` result from it. In Tcl, the returned argument for commands such as `all_inputs` or `all_clocks` is a collection. Unfortunately, Design Compiler or PrimeTime has no single command to add or subtract collections from each other. Instead, one must operate on the elements within a collection individually and work with each element to add or subtract into another collection using the `add_to_collection` or `remove_from_collection` commands.

Doing the dsch example with Tcl in Design Compiler or Primetime isn't as easy since both returned values are collections and now we have to remove a collection from a collection. Here would be the equivalent commands in Tcl:

```
set in_list [all_inputs]
foreach_in_collection element [all_clocks] {
    set in_list [remove_from_collection $in_list [get_object_name $element]]
}
```

Removing a Collection Within a Collection

In the example above, the `in_list` variable set by the `all_inputs` command is really a collection, not a list. The second command iterates over the collection returned by the `all_clocks` and allows access to each item within the collection. Note that each item in the collection, referenced by the `element` variable, is also a collection. The script then removes that collection item from the `in_list` using the `remove_from_collection` command, accomplishing the removal of the clock pins from our input pins of the design.

Programming tip: Use the `add_to_collection` and `remove_from_collection` commands when adding or removing items within a collection. Collections are not lists!

4.5 Synopsys-specific Commands Which Cause Tcl Torture

In 2000.05, setting variables on the command line of `dc_shell` using the `-x` facility must be Tcl-based, rather than DCSH-based. For example:

```
dc_shell -x "G_LIB_NAME=lsi10k"
```

Supported in 1999.10

```
dc_shell -x "set G_LIB_NAME lsi10k"
```

Newer method for 2000.05

Another item that deviates from the Tcl language is the use of UNIX style arrow redirection. Synopsys versions past 1999.10 do not support redirection. Instead, the `redirect` command must be used to pass any Synopsys specific commands into a file.

<pre>report_cell > cell.rpt</pre>	<pre>redirect cell.rpt {report_cell}</pre>
Supported in 1999.10	Newer method for 2000.05

Here is an example of using `redirect` to a null device similar to doing a “`> /dev/null`” in UNIX.

```
global G_TOPS;
set G_TOPS(NULL) {/dev/null};          # add elem NULL to global array
set G_TOPS(TMP) {/tmp/tops};          # add elem TMP to global array
...
# If port does not exist, send warning results to bit bucket
redirect $G_TOPS(NULL) \
{set test4rst [get_ports -filter {@port_direction == in} $reset_name]};
```

Using the `redirect` Command

In this example, a global variable called `G_TOPS` (we use a `G_` prefix to signify a global variable which aids in debugging and code readability) is set as an array with the `NULL` element set to `/dev/null` and the `TMP` element pointing to `/tmp/tops`. The `redirect` command puts the output caused by the `set` command of the results of the sub-executed `get_ports` command to `/dev/null`.

Getting the licenses used causes some extra Tcl code due to a feature in Synopsys 1999.10 and 2000.05 where the `list_licenses` procedure returns a text string that cannot be set into a variable. Instead, some torture in Tcl code is required:

```
exec touch [eval pid].tmp;
exec rm -rf [eval pid].tmp;
redirect [eval pid].tmp {list_licenses};
set licenses [exec cat [eval pid].tmp];
exec rm -rf [eval pid].tmp;
```

exec, `redirect` and `eval` Example

In the above example, a temporary file is created using the current process ID (`pid`) of the `dc_shell` program. The temporary file is first touched, then removed, and then set with the output of the `list_licenses` command. The `licenses` variable is then set with the contents of the temporary file. Finally, the temporary file is removed. Note that there is a double-nested bracket declaration for the `eval` and the external `cat` commands.

Programming tip: discontinue any redirect usage using the UNIX-style `>` command. It is not supported in 2000.05 and beyond. Some commands aren't variable-friendly, so redirection to a temporary file and then back into a variable may be required to get Synopsys command output into a variable.

5.0 Items For Enhancement

Here's list of some items that could be fixed or added to the Tcl interface that would aid debugging and synthesis operations.

1. Fix commands like `list_licenses`, `report_constraint`, etc to return string values that can be assigned to a variable rather than text output to the Design Compiler console. To get around this problem the Tcl-tortured user must use the `redirect` command to stuff the Synopsys (broken) command output to a temporary file and then suck the output back into a variable from the file. All this could be fixed by allowing the output to be set to a Tcl variable.
2. A built-in Tcl debugger. Design Compiler and Primetime needs a command line driven debugger much like the Perl debugger. A nice graphical debugger in Design Vision would be another good idea. The ability to single-step through sections of code and evaluate variables would be extremely helpful to overcome Tcl torture.
3. Inclusion of Tk into the Design Compiler application. Custom GUI's or enhanced API's would be a huge benefit for monitoring and control of synthesis jobs. Synopsys did not integrate Tk into the Design Compiler application which is somewhat like providing a great bottle of wine without a decent wine glass to put it in: you can drink out of the bottle, but it isn't very elegant and the etiquette value is horrible. The Tcl language is not complete without Tk!
4. Ability to show commands being executed from a procedure in the Design Compiler console. Commands executed in a Tcl script or through a source command show up, but they don't in a procedure. This hinders the ability to debug or keep a log of commands executed.
5. Ability to easily add or subtract collection elements, such as the following syntax:

```
in_list = expr [ [all_inputs] - [all_clocks] ]
```

This would be make programming constructs much easier and eliminate a lot of Tcl code to iterate or index through the collections.

6. Adding a `-example` option to the `define_proc_attributes` command. Only one `-info` option isn't enough for complete documentation of a procedure.

6.0 Useful Miscellaneous Tcl Examples

The TOPS synthesis environment uses many procedures and operations to create an easy, powerful, synthesis environment that can be easily configured to meet the project requirements. Here are some useful examples extracted from the TOPS Tcl code.

6.1 Including Procedures and Scripts

In the main TOPS Tcl script, there is the requirement to source particular Tcl code if the script exists in the specified search path. This option is used for over-riding the main functions, such as setting up clocks or changing compile strategies, on the design unit being synthesized. If a compile script exists locally, use that one rather than the main compile strategy. To accomplish this, we utilized a procedure to check if the script exists and then source the script if it is found. The procedure also prints a timestamp and the invoking script's name to the console that is useful for debugging or history logging.

```
#####
# Usage: P_source_if_exists filename caller
#
# This procedure is used to see if a file exists.  If it doesn't, return a zero.
# caller added to print invoking script's name
#####
proc P_source_if_exists {filename caller} {
    if {[which $filename] != ""} {
        set LOCAL_TIME_MARK [clock seconds];
        set full_name [which $filename];
        echo "#$caller: Sourcing $full_name";
        # Source the file in the top-level context
        uplevel source $filename;
        return 1;
    } else {
        # File was not found
        return 0;
    }
}; # end P_source_if_exists
```

P_source_if_exists Procedure

6.2 Get License Procedure

To be user friendly and not consume licenses unnecessarily, this procedure is used to check out specific licenses. The procedure will wait for up to an hour, checking at 60 second intervals if a license cannot be obtained. This procedure also fixes a flaw in Design Compiler where the `get_license` command will return an error status if Design Compiler already has the license.

```
#####
# Usage: P_get_license HDL-Compiler
#
# This procedure grabs the specified license.  Note that DC gives error
# status if we already have the license.  Hence, must determine if we
# do have the license first before we actually get it.
# Also re-check for the license in 60 second intervals for
# 1 hour before we exit with an error if we can't get a license.
#####
proc P_get_license {ln} {
```

```

set fetch_license 1;
set sleep_val 60;          # sleep in seconds we wait for a license
set max_timeout 60;       # number of 60-second waits until we die
# First must determine which licenses we have.  To do this
# we get the output of list_licenses into a TCL variable.
#
# Hack! DC is broke & can't set list_lic output to a variable,
# so must stuff it to a tmp file & read it back in.  Use process
# id (pid) in filename as a safer-alternative, touch & rm -rf to
# first to be extra safe.  Bad Synopsys!  Bad Synopsys!
exec touch [eval pid].tmp;
exec rm -rf [eval pid].tmp;
redirect [eval pid].tmp {list_licenses};
set licenses [exec cat [eval pid].tmp];
exec rm -rf [eval pid].tmp;

# now that we have a TCL variable with the list_license output,
# strip out the crud that DC uselessly puts in and parse the list
regsub {Licenses in use:} $licenses {} licenses;
foreach checkedout_license $licenses {
    if [string match $checkedout_license $ln] {
        set fetch_license 0;
    }
}
if {$fetch_license == 1} {
    set sleep_time 0;
    set dc_status [get_license $ln];
    while { ($dc_status == 0) && ($sleep_time < $max_timeout) } {
        if {$sleep_time == 0} {
            set current_time [exec date];
            echo "#INFO: waiting for license $ln @ $current_time";
        }
        sh sleep $sleep_val;
        set sleep_time [expr $sleep_time + 1];
        set dc_status [get_license $ln];
    }
    if {$dc_status == 0} {
        echo "Error: Cannot get license $ln after $sleep_time seconds, dying!";
        return 0;
    } else {
        #echo "#INFO: checked out license $ln";
        return 1;
    }
} else {
    #echo "#INFO: already have license, continuing along";
    return 1;
}
}; # end P_get_license

```

The P_get_license Procedure

6.3 Reading Verilog, VHDL, and Sub-Functional Block Procedures

Our design environment is a mix of Verilog and VHDL. We also utilize top-down as well as bottom-up compiling strategies. To minimize the license usage for both languages and deal with top-down compiling strategies, these procedures are used.

```

proc P_read_vhdl {filename {libname default}} {

```

```

P_get_license "VHDL-Compiler";
if {$libname == "default"} {
    analyze -f vhdl $filename;
} else {
    analyze -library $libname -f vhdl $filename;
}
remove_lic "VHDL-Compiler";
}; # end P_read_vhdl;

```

The P_read_vhdl Procedure

```

proc P_read_verilog {filename {libname default}} {
    P_get_lic "HDL-Compiler";
    if {$libname == "default"} {
        analyze -f verilog $filename;
    } else {
        analyze -library $libname -f verilog $filename;
    }
    remove_lic "HDL-Compiler";
}; # end P_read_verilog;

```

The P_read_verilog Procedure

```

proc P_read_subfubs {subfubs {libname default}} {
    global G_SRC_PATH;
    # Check if any subfubs to read. If none, exit else process list;
    if {![string match $subfubs ""]} {
        set subfub_list [join $subfubs];      # must join it into list first;
        echo "SUBFUBS: $subfub_list";
        # Now parse out each file;
        foreach {subfub} $subfub_list {
            set dirpath [file dirname $subfub];      # get directory path first;
            # No directory, use $G_SRC_PATH
            if {[string match $dirpath "" ] == 1} {
                set subfub [file join $G_SRC_PATH $subfub];
            }
            switch -exact -- [file extension $subfub] {
                .vhd {
                    if {$libname == "default"} {
                        P_read_vhdl $subfub;
                    } else {
                        P_read_vhdl $subfub $libname;
                    }
                }
                .vhdl {
                    if {$libname == "default"} {
                        P_read_vhdl $subfub;
                    } else {
                        P_read_vhdl $subfub $libname;
                    }
                }
                .v {
                    if {$libname == "default"} {
                        P_read_verilog $subfub;
                    } else {
                        P_read_verilog $subfub $libname;
                    }
                }
                .db {

```

```

        echo "#INFO: Reading db $subfub";
        read_db $subfub;
    }
    default {
        echo "#INFO: Reading db $subfub";
        read_db $subfub;
    }
}; # end switch
}; # end foreach
}; # end if
}; # end P_read_subfubs;

```

The P_read_subfubs Procedure

6.4 The Insert Buffer Script

On a design project, we had to fix hold violations manually rather than letting Synopsys go through an incremental compile with the `set_fix_hold` command. We accomplished this task with a procedure to find any hold time violations using the `report_constraint` command, get all end-point violations, and then add in a cell of the user's choosing (usually a buffer) to the end of the violated path. The procedure requires the user to pass in the cell type to be inserted, an instance name to call the cell, and the place to be inserted (scan path, data path, or both paths). This obviously isn't the preferred methodology to fix hold time violations, but it worked well to get the design out of the proverbial fire and provided a good example of the power of using Tcl in synthesis. Credit goes to Doug Hergatt and Tim Wilson for the script example.

```

#####
# Usage: P_proj_insert_buffers lib_cell_name inst_string fix_hold_mode
#
# This procedure is used to fix HOLD violations by inserting strategic
# buffers in the scan, data or both paths of the design. It is intended
# to be invoked pre-layout to reduce the number of violations.
#####

proc P_proj_insert_buffers {lib_cell_name inst_string fix_hold_mode} {

    # Generate list of violators & filter on VIOLATED
    # Hack! Must dump this to a tmp file because the report_constraint
    # command output can't be set to a variable! Bad Synopsys! Bad Synopsys!
    redirect tmp { report_constraint -min_delay -all };
    set violator_list [exec sed -n -e "/VIOLATED/p" tmp];
    sh rm tmp;

    # Check for no violations
    if {$violator_list == ""} {
        echo "#INFO: No MIN violations in current design";
        return;
    }; # end if

    set si_violator_list "";
    set d_violator_list "";

    # Now filter on "/SI" & create list
    foreach violator $violator_list {
        if [string match */SI $violator] {
            set si_violator_list [concat $si_violator_list $violator];
        }; # end if
    }
}

```

```

    if [string match */D $violator] {
        set d_violator_list [concat $d_violator_list $violator];
    }; # end if
}; # end foreach

# Build the list to fix
if {$fix_hold_mode == "scan"} {
    set all_violator_list $si_violator_list;
} elseif {$fix_hold_mode == "data"} {
    set all_violator_list $d_violator_list;
} elseif {$fix_hold_mode == "both"} {
    set all_violator_list [concat $d_violator_list $si_violator_list];
} else {
    echo "ERROR: Incorrect Fix Hold Mode. Please specify scan, data or both";
}; # end if

# Reset cell counter (used to create a unique instance name
set cell_cnt 0;

echo "#INFO: Fixing Hold violations on the violator paths.";

# Fix all the violations for the violator list
foreach reg_cell $all_violator_list {

    # Get the instance name of the violator
    regexp {(.*).*.} $reg_cell var1 var2
    set inst_name [file tail $var2];

    # Get the hierarchy path to the violating register
    regexp {(.*).*.} $var2 var1 hier_path

    # Increment the cell counter
    set cell_cnt [expr $cell_cnt + 1];

    # Find everything connected to the pin of this register
    set orig_net [all_connected [get_pins $reg_cell]];
    # Create new instance name (HOLD delay cell)
    set new_cell ${inst_name}${inst_string}${cell_cnt};
    # Create new net name (net inserted between new HOLD cell & register pin)
    set new_net ${new_cell}_net;

    # Disconnect the orig net & create the new cell & net
    disconnect_net $orig_net $reg_cell;
    create_cell -instance $hier_path $new_cell $lib_cell_name;
    create_net -instance $hier_path $new_net;

    # Add the hierarchy path to the new net & cell vars
    set new_net [file join $hier_path $new_net];
    set new_cell_in [file join $hier_path ${new_cell}/A];
    set new_cell_out [file join $hier_path ${new_cell}/O];

    # Connect the nets
    connect_net $new_net $reg_cell;
    connect_net $new_net $new_cell_out;
    connect_net $orig_net $new_cell_in;
}; # end foreach

echo "#INFO: Added $cell_cnt HOLD cells to the design.";
}; # end proc

# document the procedure!

```

```

define_proc_attributes P_proj_insert_buffers \
  -info "PROJ: Procedure to fix HOLD violations inserting strategic buffers" \
  -define_args {
    {lib_cell_name "Library Cell Name to insert" lib_cell_name string required}
    {inst_string "Instance String to insert in the new cell" inst_string string
required}
    {fix_hold_mode "Fix Hold Mode: scan, data, or both" fix_hold_mode string
required}
  }

```

P_proj_insert_buffers Script Example

7.0 Conclusion

Tcl is a very powerful language that enhances the synthesis tasks. Tcl does have quirks and oddities in its usage and syntax that conflict with more mainstream programming languages. In addition, the Synopsys-specific Tcl features create some confusion in implementing Tcl. This paper hopefully has shown some of the major pitfalls that even experienced programmers face and it has provided real-world example code.

The TOPS synthesis environment was a joint effort by Gregg Lahti, Steve Brown, Tim Wilson, Rodney Pesavento, and Doug Hergatt (CX Design). This paper couldn't have been accomplished without the monster-sized effort from Doug. Doug's expertise in design and synthesis provided great insight into solving many of the basic problems faced when the TOPS synthesis environment was created. He was also the voice of sanity and reason when we couldn't agree on methodology or direction.

8.0 Reference

Ousterhout, John K., Tcl and the Tk Toolkit, Addison-Wesley, 1994. ISBN: 020163337X

Welch, Brent B., Practical Programming in Tcl & Tk, Second Edition, Prentice Hall, 1997. ISBN 0-13616830-2

Nelson, Christopher, Tcl/Tk Programmer's Reference, Osbourne/McGraw Hill, 2000. ISBN 0-07-212004-5

Tcl Developer Exchange Website: <http://ajubasolutions.com>

Synopsys Solvnet Website: <http://solvnet.synopsys.com/cgi-bin/ASP/solvnet/sign-on>

Wilson, Tim L, and Pesavento, Rodney, Using Tcl to Implement an Efficient Synthesis Environment (TOPS), Boston Synopsys Users Group Conference, September 2000