

Title Page

# **My Favorite DC/PT TCL Tricks**

Paul Zimmer

Zimmer Design Services  
1375 Sun Tree Dr.  
Roseville, CA 95661

[paulzimmer@zimmerdesignservices.com](mailto:paulzimmer@zimmerdesignservices.com)

## ABSTRACT

Tcl is rapidly becoming the dominant scripting language for DesignCompiler and PrimeTime. This paper will show some of the author's favorite dc/pt Tcl tricks and techniques.

For example, subroutines (procedures) are a critical tool in automating and maintaining your Tcl scripts. But argument parsing and so forth can be a pain. You end up spending more time mucking with the overhead stuff than working on the real guts of the routine.

This paper will show you how to use Synopsys' built-in tools, plus a few of the author's own helpful procs, to streamline the more mundane aspect of creating and maintaining your own procs.

Have you ever been frustrated because you have an error in your script, but can't figure out where because the tool won't echo your commands as it executes them? This paper will show you how to fix this tool oversight.

Various other tips and tricks will be covered as well.

## Table of contents

1	Introduction .....	4
2	Building better procs using parse_proc_arguments and define_proc_attributes.....	4
2.1	The basics.....	4
2.2	Adding an optional argument .....	7
2.3	Cleaning up the \$results(option) mess.....	9
2.4	Getting proc name with &myname .....	13
2.5	Putting the default values where they belong.....	15
2.6	Putting it all together .....	21
3	Following the flow with &cmd .....	23
3.1	The problem – can’t follow the execution flow.....	23
3.2	The solution – use a wrapper .....	25
3.3	&list_coll.....	26
3.4	&cmd .....	28
4	Creating notifier force files from Primetime .....	30
4.1	The problem – X’s from sync’ing flops create chaos in simulation.....	30
4.2	The solution – force the notifiers.....	30
4.3	Creating the notifier force file from PrimeTime .....	31
4.4	&create_force_file .....	33
4.5	&create_force_on_collection.....	34
5	More Fun Stuff .....	41
5.1	Defaulting variables using &default.....	41
5.2	Using suppress_message and unsuppress_message to control warning messages.....	42
6	Conclusion .....	44
7	Acknowledgements .....	44
8	References .....	44
9	Appendix .....	45
9.1	Perl script to create _refnames2notifiers.....	45

## 1 Introduction

Way back in the early days of Synopsys, `dc_shell` (there was no PrimeTime back then) didn't use Tcl. Instead, it had its own weird shell language. This language was extremely awkward to use, having no subroutines and only simple variables and lists. The lists couldn't even be accessed by index. Life was hard.

Into this mess stepped a consultant by the name of Steve Golson. Steve gave a presentation at SNUG (San Jose - 1995) called "My Favorite `dc_shell` Tricks". This opened up a whole new world for me. Steve had found ways to fake out subroutines using aliases, create complex data structures using only the simple lists, and many other things that made the language a bit more usable, if still somewhat difficult to read and maintain. Over the years, I built a whole complex synthesis environment using Steve's techniques.

In the last 5 years, I have been specializing more and more in synthesis and static timing. As a result, I have had the opportunity to explore the Synopsys tools and their Tcl interface in more detail than most users can afford. In the process, I have discovered some things and developed some techniques that may be useful to others.

This paper is my attempt to return the favor to Steve - and to the user community. I don't expect anything here to have the "life changing" value of Steve's long-ago paper, but I hope people find things they can use to make their own Synopsys coding easier.

So, in a tribute to "My Favorite `dc_shell` Tricks", here are my own favorite `dc/pt` Tcl tricks.

## 2 Building better procs using `parse_proc_arguments` and `define_proc_attributes`

Most people end up using procs (Tcl subroutines) to structure their code, but many people don't realize that the Synopsys extensions to Tcl in `dc_shell` and `pt_shell` have built-in tools to make the job of creating and maintaining your own procs much easier. These tools provide automatic parsing of the command arguments, automatic validation of the command arguments, and hooks into the help command that make your proc's return help information just like Synopsys' own commands. Taken together, these tools can make building, maintaining, and using your own procs much easier.

### 2.1 The basics

The basic tool set is comprised of two commands – `parse_proc_arguments` and `define_proc_attributes`. Using these commands is simple. You just declare your proc with only a single keyword as the argument ("args") like this:

```
proc myproc args {
```

Then, inside the proc, you make a call to “parse\_proc\_arguments”, giving it the single argument from the declaration line (“args”) and the name of the return array (traditionally called “results”):

```
parse_proc_arguments -args $args results
```

Outside the proc (but before you first use it) you define your arguments, what type they are, whether they are optional or required, etc using the command “define\_proc\_attributes”:

```
define_proc_attributes myproc -info "Description" \  
-define_args \  
{  
  {arg_description}  
  {arg_description}  
  ...  
}
```

As an example, here is a little proc called &dump\_coll<sup>1</sup> that dumps the object names in a collection, one item per line (like “query collection” but one per line):

```
proc &dump_coll args {  
  # Call the standard parser  
  parse_proc_arguments -args $args results  
  
  # Body of process  
  foreach_in_collection _item $results(_collection) {  
    echo [get_object_name $_item]  
  }  
}  
define_proc_attributes &dump_coll -info "Dump a collection formatted one per  
line" \  
-define_args \  
{  
  {_collection "Collection" target_collection list required}  
}
```

The use of parse\_proc\_arguments is pretty obvious, but let’s look at define\_proc\_attributes in a little more detail.

The first argument to define\_proc\_attributes is the name of the proc whose attributes are being described – in this case “&dump\_coll”. The “-info” option is followed by a string that will be used as the proc’s description by the help command (more on help later). The heart of the command is the “-define\_args” option. It is a list of lists that define each argument to your command and the properties of that argument. In the example above, we are describing a required argument called “collection” whose value is a list (collections are considered lists by define\_proc\_attributes) .

---

<sup>1</sup> coming from perl background, I like my subroutine names to start with “&”

Here are the details of what the fields mean:

- The first field is what the man page calls the argument name. It is essentially where to store the value after parsing. When `parse_proc_arguments` is called from within the procedure, it will return an associative array (traditionally called “results”) with all of the argument values given when the proc was invoked. This field is the index into that array. So, after `parse_proc_arguments` is called, there will be an entry “results(`_collection`)” set to the collection specified by the user when `&dump_coll` was called.
- The second field provides a description of the argument for `help -v`
- The third field provides the name of the argument for `help -v`. Use a null string (“”) for boolean arguments, as the `help -v` will use the first field as the name for booleans. See the example below.
- The fourth field gives the expected data type. This can be: string, list, boolean, int, float, or `one_of_string`.
- The fifth field can be rather complicated, but for simple examples it usually just specifies whether the argument is optional or required. Combined with `one-of-string` in the fourth field, it can also do some clever stuff with allowing only values from a specified set. See the man page for details.

Note that the first and third fields are traditionally given the same value. I have made them different here to illustrate what each field really does.

So, let’s load up `&dump_coll` and see what we’ve got:

```
dc_shell-t> source dc_basic.dctl
dc_shell-t> q [get_cells *]
{"doutpad", "clktree", "clkoutpad", "dout_reg"}
dc_shell-t> &dump_coll [get_cells *]
doutpad
clktree
clkoutpad
dout_reg
```

So, `dump_coll` did what it was supposed to do and dumped the names of the items in the collection “[`get_cells *`]”, one per line.

Now let’s see what `help` says about `&dump_coll`:

```
dc_shell-t> help &dump_coll
&dump_coll          # Dump a collection formatted one per line
```

Pretty slick. Looks just like a built-in command. How about help -v?

```
dc_shell-t> help -v &dump_coll
&dump_coll          # Dump a collection formatted one per line
  target_collection  (Collection)
```

How about argument validation? Let's see what happens if we fail to give it the required argument, or give it an argument of the wrong type:

```
dc_shell-t> &dump_coll
Error: Required argument '_collection' was not found (CMD-007)
dc_shell-t> &dump_coll 1.0
Error: No such collection '1.0' (SEL-001)
```

So, we got argument parsing, validation, and help information for just a few simple lines of code. Kuhl. Thanks, Synopsys.

## 2.2 Adding an optional argument

Now let's add an optional argument. You wouldn't want &dump\_coll to output a million items, so some sort of limit is required in the loop. Let's make this limit a proc option. To do this, we need to do two things. First, we need to add this to the argument definitions in define\_proc\_attributes:

```
{-limit "Max to dump (default 100)" limit int optional}
```

This tells define\_proc\_attributes that an optional argument called “-limit” has been added, and that the expected data type is “int”.

Second, we need to default the “results(-limit)” entry to handle the case where the option is not used (parse\_proc\_arguments doesn't do this for us – more on this later). The easiest way to do this is to set the results entry to the default value before calling parse\_proc\_arguments:

```
set results(-limit) 100
```

Finally, we need to add the actual loop count code. Here's the new &dump\_coll proc:

```

proc &dump_coll args {
    # Default optional arguments
    set results(-limit) 100

    # Call the standard parser
    parse_proc_arguments -args $args results

    # Body of process
    set _count 0
    foreach_in_collection _item $results(_collection) {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $results(-limit)} {
            echo "... hit limit $results(-limit), stopping ..."
            break;
        }
    }
}
define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {_collection "Collection" target_collection list required}
        {-limit "Max to dump (default 100)" limit int optional}
    }
}

```

Naturally, my real code would be chock-full of helpful comments about how this all works, but I'm trying to keep things compact (ahem).

So, let's try &dump\_coll with and without the new limit option:

```

dc_shell-t> &dump_coll [get_cells *]
doutpad
clktree
clkoutpad
dout_reg

dc_shell-t> &dump_coll -limit 2 [get_cells *]
doutpad
clktree
... hit limit 2, stopping ...

```

How about the help -v output?

```

dc_shell-t> help -v &dump_coll
&dump_coll          # Dump a collection formatted one per line
  [-limit limit]    (Max to dump (default 100))
  target_collection (Collection)

```

There's our new -limit option. Pretty easy.

## 2.3 Cleaning up the \$results(option) mess

That's really nice, but accessing the argument values using "\$results(blah)" all the time gets really old. What we'd like is to just stick them in simple variables. I like all my variables to start with "\_"<sup>2</sup>, so I want my argument values to end up in variables called "\_collection" and "\_limit".

I could just do:

```
set _collection $results(_collection)
set _limit $results(-limit)
```

But that means I have to remember to do this manually every time I add an argument. It's easy enough to do this in a generic loop:

```
foreach _argname [array names results] {
  regsub -- {-} $_argname {_} _varname
  set $_varname $results($_argname)
}
```

The foreach loops over all the entries in results. The first line of the loop substitutes "--" with "\_". I cleverly give my required arguments names that already start with "\_" (that's why the target\_collection argument definition starts with "\_collection" rather than "collection"), so I only have to change the "--" arguments.

---

<sup>2</sup> I think I got this technique from Steve Golson as well. The idea is to avoid accidentally stepping on DC/PT built-in variables. Unfortunately, Synopsys isn't consistent about their own variable naming, and running "print\_variable\_group all" in dc\_shell-t produces the following built-in variables beginning with "\_":

```
_Variable_Groups = "<array?>"
__err = "can't read "env(LMC_HOME)": no such variable"
_acs_internal_inside_ACS = "false"
```

And, of course, collection handles are "\_sel...".

The second line of the loop just sets the new variable value. Pretty simple. Here's our `&dump_coll` proc now:

```
proc &dump_coll args {
    # Default optional arguments
    set results(-limit) 100

    # Call the standard parser
    parse_proc_arguments -args $args results

    foreach _argname [array names results] {
        regsub -- {-} $_argname {} _varname
        set $_varname $results($_argname)
    }

    # Body of process
    set _count 0
    foreach_in_collection _item $_collection {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $_limit} {
            echo "... hit limit $_limit, stopping ..."
            break;
        }
    }
}
define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {_collection "Collection" target_collection list required}
        {-limit "Max to dump (default 100)" limit int optional}
    }
}
```

Notice that all the references to “`$results(blah)`” have been replaced with direct references to the variables “`_collection`” and “`_limit`”.

Not content to cut and paste this loop code around, now I want to roll the loop code into a proc itself. I'll call this new proc “`&fixargs`”<sup>3</sup>.

```
proc &fixargs {} {
    # Link to results of calling procedure
    upvar results results
    # Put values in variables with argument names (sub - with _)
    foreach _argname [array names results] {
        regsub -- {-} $_argname {} _varname
        set _cmd "set $_varname \{${results($_argname)}\}"
        uplevel 1 $_cmd
    }
}
```

---

<sup>3</sup> This is not to imply that the arguments are broken, but rather that the proc is going to “fix them up”.

This is similar to the inline code above. To get access to the results array, we use “upvar” to link the local results array to that of the calling procedure. Then we loop through the elements of \$results as before, except that instead of just executing the set command directly, we put it in a variable (\$\_cmd), then execute it using “uplevel”. This causes the command to be executed in the context of the calling procedure.

I find that many of my proc’s end up with a “-verbose” option. Rather than have to default this using “set results(-verbose) 0” all the time, I’ll just default it in &fixargs. Also, I often find that I want to just turn on the -verbose switch in all my procs using a single global variable rather than have to edit each proc call. I’ll integrate this into &fixargs, too.

```
proc &fixargs {} {
    global _global_verbose

    # Link to results of calling procedure
    upvar results results
    # Put values in variables with argument names (sub - with _)
    foreach _argname [array names results] {
        regsub -- {-} $_argname {} _varname
        set _cmd "set $_varname \${results($_argname)}\"
        uplevel 1 $_cmd
    }

    # Special handling for _verbose...
    # If it doesn't exist (in proc above), create it and set it to 0
    set _cmd "info exists _verbose"
    if ![uplevel 1 $_cmd] {
        set _cmd "set _verbose 0"
        uplevel 1 $_cmd
    }
    # If global_verbose is set, turn on _verbose in level above (harmless if
    # not used)
    if {[info exists _global_verbose] && $_global_verbose} {
        set _cmd "set _verbose 1"
        uplevel 1 $_cmd
    }
}
```

The new code just checks for the existence of verbose in the calling procedure, and sets it to 0 if it doesn’t already exist. This allows proc’s to default \_verbose to 1. Then, if the “\_global\_verbose” variable exists and is set (note that it is declared as a global at the start of the procedure), then the code sets \_verbose in the calling procedure to 1.

Here's what `&dump_coll` looks like now:

```
proc &dump_coll args {
    # Default optional arguments
    set results(-limit) 100

    # Call the standard parser
    parse_proc_arguments -args $args results

    &fixargs

    # Body of process
    set _count 0
    foreach_in_collection _item $_collection {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $_limit} {
            if {$_verbose} {
                echo "... hit limit $_limit, stopping ..."
            }
            break;
        }
    }
}
define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {_collection "Collection" target_collection list required}
        {-limit "Max to dump (default 100)" limit int optional}
        {-verbose "Be Chatty" "" boolean optional}
    }
}
```

I have added the `-verbose` option to `define_proc_attributes`, and used it to enable the “hit limit” message. This isn’t necessarily the way you would want the real proc to work (you probably want the “hit limit” message to come out whenever you hit the limit), but it’s handy for illustration.

So, now let's explore some of the new features:

```
dc_shell-t> &dump_coll [get_cells *]
doutpad
clktree
clkoutpad
dout_reg

dc_shell-t> &dump_coll -limit 2 [get_cells *]
doutpad
clktree

dc_shell-t> &dump_coll -verbose -limit 2 [get_cells *]
doutpad
clktree
... hit limit 2, stopping ...

dc_shell-t> set _global_verbose 1
Information: Defining new variable '_global_verbose'. (CMD-041)
1

dc_shell-t> &dump_coll -limit 2 [get_cells *]
doutpad
clktree
... hit limit 2, stopping ...
dc_shell-t> help -v &dump_coll
&dump_coll          # Dump a collection formatted one per line
  [-limit limit]      (Max to dump (default 100))
  [-verbose]          (Be Chatty)
  target_collection   (Collection)
```

## 2.4 Getting proc name with &myname

Here's another cool feature we can add. It doesn't matter much for a small, simple proc like &dump\_coll, but for larger, more complex procs which call other procs, I sometimes find it difficult to figure out which proc is executing. The simplest solution to this is just to add a message at the beginning and end of the proc giving the proc name and whether it is starting or ending. We could just use "echo Starting &dump\_coll" and "echo End of &dump\_coll", but I find it tedious keeping these up-to-date as I cut and paste and create new procs from old ones. So, let's find an automated way to do this.

First, we need to find a way to get the name of the procedure we're in. I tried the usual \$0, but that didn't work. In the end, I fell back on the "info" command. It turns out that "info level" returns the current call stack level, but "info level N" (where N is a positive integer) returns the full invocation line of the procedure at level N. So, "info level [info level]" would return the name and arguments of the current procedure. Since I want to put this code in a procedure itself, I need to do "info level [expr [info level] - 1]":

```
proc &myname { } {
  set _my_level [info level]
  set _caller_level [expr $_my_level -1]
  set _caller_info [info level $_caller_level]
```

This returns a list whose first element is the name of the procedure. I need to split this list into an array, then extract the “0<sup>th</sup>” element, like this:

```
proc &myname { } {
    set _my_level [info level]
    set _caller_level [expr $_my_level -1]
    set _caller_info [info level $_caller_level]
    set _array [split $_caller_info]
    return [lindex $_array 0]
}
```

Compressing it, our &myname procedure turns out to be a “one liner”:

```
proc &myname { } {
    return [lindex [split [info level [expr [info level] -1]]] 0]
}
```

So, here’s our latest &dump\_coll:

```
proc &dump_coll args {

    # Default optional arguments
    set results(-limit) 100

    # Call the standard parser
    parse_proc_arguments -args $args results

    &fixargs

    # Body of process
    echo "Starting [&myname]"
    set _count 0
    foreach_in_collection _item $_collection {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $_limit} {
            if {$_verbose} {
                echo "... hit limit $_limit, stopping ..."
            }
            break;
        }
    }
    echo "End of [&myname]"
}
define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {_collection "Collection" target_collection list required}
        {-limit "Max to dump (default 100)" limit int optional}
        {-verbose "Be Chatty" "" boolean optional}
    }
}
```

## 2.5 Putting the default values where they belong

The last remaining bit of clumsiness is the need to default the `$results()` entries before calling `parse_proc_arguments`. This is a real nuisance. Every time I add a new option, I have to remember to go back and insert this code. To my mind, this is really a shortcoming in `define_proc_attributes`. It should really have a default value field in the `-define_args`.

WARNING. All the rest of the code in this paper is well-seasoned and production-tested. The following section is nearly virgin code. Proceed with caution.

The “sweet” (sorry, I have teenage kids) way around this is to put `define_proc_attributes` itself in a wrapper, parse the new field, and create entries in a global array to specify the default values. This array is then accessed by a proc called just before `parse_proc_arguments`, or better still, `parse_proc_arguments` can be put in a wrapper of its own.

We have two basic approaches to choose from when creating the wrapper version of `define_proc_attributes` (henceforth called `&define_proc_attributes`). We could treat it as a normal proc and use all the processing tricks discussed so far. However, this requires duplicating the options to `define_proc_attributes` in a standard call to `define_proc_attributes`. This would then require tracking any changes that Synopsys makes to `define_proc_attributes`.

The other approach is to implement `&define_proc_attributes` as a “simple” proc, and not use `parse_proc_arguments` and `define_proc_attributes`. That is the approach I have taken here, partly to avoid the need to track changes to the standard `define_proc_attributes` command, and partly to illustrate the basic “wrapper” technique.

First, the big picture. This command is going to be used exactly like the standard `define_proc_attributes` command, except that there will now be a sixth field for optional arguments. The value in this field will be parsed, and made available to the proc being defined so that it can set its defaults. The way I’m going to pass this info is via a global associative array indexed by the proc name, with the data being a series of “set” commands that implements the default values.

In addition, any “boolean” options will automatically get defaulted to 0.

Here’s an example using our old friend `&dump_coll`. I have changed the “-verbose” to “-chatty” to better illustrate boolean defaults (because “-verbose” gets defaulted by `&fixargs`):

```
&define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
  -define_args \
  {
    {_collection "Collection" target_collection list required}
    {-limit "Max to dump" limit int optional 2}
    {-chatty "Be Chatty" "" boolean optional}
  }
```

Note that I have set the default limit to “2” (the “2” following “optional” on the 5<sup>th</sup> line).

After `&define_proc_attributes` runs, I should have an entry in my associative array (called `_proc_defaults`) like this:

```
dc_shell-t> echo $_proc_defaults(&dump_coll)
set results(-chatty) { 0 } ; set results(-limit) { 2 } ;
```

With that understood, here is `&define_proc_attributes`:

```
proc &define_proc_attributes { args } {

    global _proc_defaults

    # Body of process

    # This is MY proc, and I'm going to insist that proc_name be first!
    set _proc_name [lindex $args 0]

    # Find arg definition field in args (follows -define_args)
    set _arg_defs_index [expr [lsearch $args {-de*}]+ 1]
    set _arg_defs [lindex $args $_arg_defs_index]

    set _cmd "" ;# init command to null
    # March through arg defs and find any optional ones with defaults
    foreach _def $_arg_defs {
        if {[lindex $_def 3] == "boolean"} {
            # Booleans always default to 0
            set _default 0
            # Now append the set to $_cmd
            set _name [lindex $_def 0]
            set _cmd "set results($_name) {_default} ; $_cmd"
        } elseif {[lindex $_def 4] == "optional" && [llength $_def] > 5} {
            # Found one. Grab default, then crush it out
            set _default [lindex $_def 5]
            set _def [lreplace $_def 5 5]
            # Modify the help comment to add default
            set _helpcomment [lindex $_def 1]
            set _helpcomment "$_helpcomment (default $_default)"
            set _def [lreplace $_def 1 1 $_helpcomment]
            # Now append the set to $_cmd
            set _name [lindex $_def 0]
            set _cmd "set results($_name) {_default} ; $_cmd"
        }
        # Rebuild the arg defs as we go
        lappend _new_arg_defs $_def
    }

    # Save away cmd in global array
    set _proc_defaults($_proc_name) $_cmd

    # Create new args by replacing arg defs with new arg defs
    set new_args [lreplace $args $_arg_defs_index $_arg_defs_index
$_new_arg_defs]

    # Go ahead and do standard define_proc_attributes using new args
    eval define_proc_attributes $new_args
}
```

```
}
```

The first line grabs the name of the proc for which `define_proc_attributes` is being run. Note that the real `define_proc_attributes` will let you put the arguments in any order. This is a hassle for me to parse (unless I use the “other” approach), so I’m going to restrict my calls to the wrapper to put the name of the proc first.

The rest of the options I can let float. So, in the next line I’m looking for “-de\*” among the args items (this is the minimum abbreviation for `-define_args`), and grabbing the next item (which is the list of lists that is the argument definition).

Having gotten my argument definitions, I will now scan them for booleans or optional arguments containing the new field. If I find the new field, I save the value for use in the “set” command creation and then get rid of it (since I don’t want the call to the real `define_proc_attributes` to see it).

As a side note, I restrict the new field to optional args because the automatic argument validation of `parse_proc_arguments` is going to insist that the field be there anyway, so there is no point in having a default value.

After all this is done, I will have a new version of the arg defs with my new field removed (called `$_new_arg_defs`). I can now do the standard “wrapper” trick of eval’ing the target command with my new arguments.

To use this feature, I have to modify `&dump_coll` to use the new `_proc_defaults` global like this:

```
proc &dump_coll args {
    global _proc_defaults

    # Default optional arguments
    eval $_proc_defaults([&myname]) ;# <- this is how I get the defaults in

    # Call the standard parser
    parse_proc_arguments -args $args results

    &fixargs

    # Body of process
    echo "Starting [&myname]"
    set _count 0
    foreach_in_collection _item $_collection {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $_limit} {
            if {$_chatty} {
                echo "... hit limit $_limit, stopping ..."
            }
            break;
        }
    }
    echo "End of [&myname]"
}
&define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {$_collection "Collection" target_collection list required}
        {-limit "Max to dump" limit int optional 2}
        {-chatty "Be Chatty" "" boolean optional}
    }
}
```

And everything still works:

```
dc_shell-t> help -v &dump_coll
&dump_coll          # Dump a collection formatted one per line
[-limit limit]      (Max to dump (default 2))
[-chatty]           (Be Chatty)
target_collection   (Collection)

dc_shell-t> &dump_coll -chatty [get_cells *]
Starting &dump_coll
doutpad
clktree
... hit limit 2 , stopping ...
End of &dump_coll
```

Notice that `&define_proc_attributes` kindly inserted the default value information “(default 2)” into the help message for the `-limit` argument.

That all looks great, but that “eval” line is a little messy. Let’s create a wrapper for `parse_proc_arguments` and roll the eval into it. This turns out to be pretty easy:

```
proc &parse_proc_arguments { args } {  
    global _proc_defaults  
  
    # Body of process  
    # Link results array  
    upvar results results  
  
    # Get name of calling proc  
    set _proc_name [uplevel 1 &myname]  
  
    # Execute default setting command (created by &define_proc_attributes  
    eval $_proc_defaults($_proc_name)  
  
    # Run standard parse_proc_arguments at level of calling procedure  
    uplevel 1 parse_proc_arguments $args  
}
```

First, we link our local results array to the one from the calling proc. Next, we use our friend `&myname`, executed up one level, to get the name of the calling proc (`&myname` alone would just return “`&parse_proc_arguments`”). Now we can do the eval of the `_proc_defaults` entry for the calling proc. Remember that this will do something like “`set results(option_name) default_value`”, meaning it changes the results array, which we had previously linked to that of the calling procedure.

Finally, we do the standard wrapper thing of eval’ing the original command. We do this using `uplevel 1` to make sure it is done in the context of the calling proc.

Now plug this into &dump\_coll:

```
proc &dump_coll args {
    global _proc_defaults

    # Call the standard parser
    &parse_proc_arguments -args $args results ;# <- now using wrapper

    &fixargs

    # Body of process
    echo "Starting [&myname]"
    set _count 0
    foreach_in_collection _item $_collection {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $_limit} {
            if {$_chatty} {
                echo "... hit limit $_limit, stopping ..."
            }
            break;
        }
    }
    echo "End of [&myname]"
}
&define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {_collection "Collection" target_collection list required}
        {-limit "Max to dump" limit int optional 2}
        {-chatty "Be Chatty" "" boolean optional}
    }
}
```

And test it:

```
dc_shell-t> help -v &dump_coll
&dump_coll          # Dump a collection formatted one per line
[-limit limit]      (Max to dump (default 2))
[-chatty]           (Be Chatty)
target_collection   (Collection)

dc_shell-t> &dump_coll -chatty [get_cells *]
Starting &dump_coll
doutpad
clktree
... hit limit 2 , stopping ...
End of &dump_coll
```

Pretty easy, huh?

## 2.6 Putting it all together

But we don't really need separate calls to `&parse_proc_arguments` and `&fixargs`. We can easily just graft `&fixargs` into `&parse_proc_arguments`:

```
proc &parse_proc_arguments { args } {

    global _proc_defaults
    global _global_verbose

    # Body of process
    # Link results array
    upvar results results

    # Get name of calling proc
    set _proc_name [uplevel 1 &myname]

    # Execute default setting command (created by &define_proc_attributes
    eval $_proc_defaults($_proc_name)

    # Run standard parse_proc_arguments at level of calling procedure
    uplevel 1 parse_proc_arguments $args

    # Now get rid of clumsy $results stuff.
    # Put values in variables with argument names (sub - with _)
    foreach _argname [array names results] {
        regsub -- {-} $_argname {} _varname
        set _cmd "set $_varname \{${results($_argname)}\}"
        uplevel 1 $_cmd
    }

    # Special handling for _verbose...
    # If it doesn't exist (in proc above), create it and set it to 0
    set _cmd "info exists _verbose"
    if ![uplevel 1 $_cmd] {
        set _cmd "set _verbose 0"
        uplevel 1 $_cmd
    }
    # If global_verbose is set, turn on _verbose in level above (harmless if
    # not used)
    if {[info exists _global_verbose] && $_global_verbose} {
        set _cmd "set _verbose 1"
        uplevel 1 $_cmd
    }
}
```

We can now take `&fixargs` out of `&dump_coll`:

```
proc &dump_coll args {
    global _proc_defaults

    # Call the standard parser
    &parse_proc_arguments -args $args results ;# <- wrapper has fixargs built-
in

    # Body of process
    echo "Starting [&myname]"
    set _count 0
    foreach_in_collection _item $_collection {
        echo [get_object_name $_item]
        set _count [expr $_count + 1];
        if {$_count >= $_limit} {
            if {$_chatty} {
                echo "... hit limit $_limit, stopping ..."
            }
            break;
        }
    }
    echo "End of [&myname]"
}
&define_proc_attributes &dump_coll -info "Dump a collection formatted one per
line" \
    -define_args \
    {
        {_collection "Collection" target_collection list required}
        {-limit "Max to dump" limit int optional 2}
        {-chatty "Be Chatty" "" boolean optional}
    }
}
```

Test it one more time:

```
dc_shell-t> help -v &dump_coll
&dump_coll          # Dump a collection formatted one per line
[-limit limit]      (Max to dump (default 2))
[-chatty]           (Be Chatty)
target_collection   (Collection)

dc_shell-t> &dump_coll -chatty [get_cells *]
Starting &dump_coll
doutpad
clktree
... hit limit 2 , stopping ...
End of &dump_coll
```

Tada! Procs made easy!

### 3 Following the flow with &cmd

#### 3.1 The problem – can't follow the execution flow

Following the execution flow in the Synopsys tools can sometimes be difficult.

Suppose we put the following sequence of commands in a file and source the file:

```
create_clock -period 10.0 [get_ports clk_in]

create_generated_clock \
  -name clkout \
  -source [get_ports clk_in] \
  -divide_by 1 \
  [get_ports clkout]

# create input constraint
set_input_delay -clock clk_in -max 9.95 in
set_input_delay -clock clk_in -min 0.1 in

# create output constraint
set_output_delay -clock clkoutt -max 9.9 dout      ;# <- typo!
set_output_delay -clock clkout -min "-1.0" dout
```

Notice the typo on the next-to-last line.

If I source this using the `-echo` switch, I get:

```
dc_shell-t> source -echo cmd_example_noif.dctl
create_clock -period 10.0 [get_ports clk_in]
create_generated_clock -name clkout -source [get_ports clk_in] -
divide_by 1 [get_ports clkout]
# create input constraint
set_input_delay -clock clk_in -max 9.95 in
set_input_delay -clock clk_in -min 0.1 in
# create output constraint
set_output_delay -clock clkoutt -max 9.9 dout
Error: Cannot find clock 'clkoutt'. (UID-250)
set_output_delay -clock clkout -min "-1.0" dout
1
```

This is fairly straightforward. I can easily see where the error is.

But suppose I put this whole sequence of commands in an if statement (or any loop or conditional):

```
set _someflag 1
if {$_someflag} {

    create_clock -period 10.0 [get_ports clkkin]

    create_generated_clock \
        -name clkout \
        -source [get_ports clkkin] \
        -divide_by 1 \
        [get_ports clkout]

    # create input constraint
    set_input_delay -clock clkkin -max 9.95 in
    set_input_delay -clock clkkin -min 0.1 in

    # create output constraint
    set_output_delay -clock clkoutt -max 9.9 dout
    set_output_delay -clock clkout -min "-1.0" dout

}
```

Now if I do the source, I get this:

```
dc_shell-t> source -echo cmd_example.dctl
set _someflag 1
if {$_someflag} {

    create_clock -period 10.0 [get_ports clkkin]

    create_generated_clock      -name clkout      -source [get_ports clkkin]
-divide_by 1      [get_ports clkout]

    # create input constraint
    set_input_delay -clock clkkin -max 9.95 in
    set_input_delay -clock clkkin -min 0.1 in

    # create output constraint
    set_output_delay -clock clkoutt -max 9.9 dout
    set_output_delay -clock clkout -min "-1.0" dout

}
Error: Cannot find clock 'clkoutt'. (UID-250)
Information: Defining new variable '_someflag'. (CMD-041)
1
```

All the commands get echoed first, then they all get executed. The “Error:” message doesn’t come out until the end, so it’s difficult to tell which line had the error. In fact, without an error, the return of an if statement with a thousand commands in it will be “1”. Not very helpful.

## 3.2 The solution – use a wrapper

I've struggled with this problem a lot over the years, and I finally hit on a solution. The solution I have found is to execute all the Synopsys “action” commands through a wrapper proc that echo's the command before executing it. I call this wrapper proc “&cmd”.

The simplest form of &cmd looks like this:

```
proc &cmd { args } {  
    # echo result  
    echo "Doing command: $args"  
  
    # Do command  
    uplevel 1 $args ;# changed 1/9/2004  
}
```

Here's what the file looks like with &cmd used:

```
set _someflag 1  
if {$_someflag} {  
  
    &cmd create_clock -period 10.0 [get_ports clkkin]  
  
    &cmd create_generated_clock \  
        -name clkout \  
        -source [get_ports clkkin] \  
        -divide_by 1 \  
        [get_ports clkout]  
  
    # create input constraint  
    &cmd set_input_delay -clock clkkin -max 9.95 in  
    &cmd set_input_delay -clock clkkin -min 0.1 in  
  
    # create output constraint  
    &cmd set_output_delay -clock clkoutt -max 9.9 dout  
    &cmd set_output_delay -clock clkout -min "-1.0" dout  
  
}
```

Now when I source the file, I get:

```
dc_shell-t> source -echo cmd_example_wcmd.dctl
set _someflag 1
if {$_someflag} {

    &cmd create_clock -period 10.0 [get_ports clkkin]

    &cmd create_generated_clock          -name clkout          -source [get_ports
clkkin]          -divide_by 1          [get_ports clkout]

    # create input constraint
    &cmd set_input_delay -clock clkkin -max 9.95 in
    &cmd set_input_delay -clock clkkin -min 0.1 in

    # create output constraint
    &cmd set_output_delay -clock clkoutt -max 9.9 dout ;# <- TYPO !
    &cmd set_output_delay -clock clkout -min "-1.0" dout

}
Doing command: create_clock -period 10.0 _sel4
Doing command: create_generated_clock -name clkout -source _sel5 -divide_by 1
_sel6
Doing command: set_input_delay -clock clkkin -max 9.95 in
Doing command: set_input_delay -clock clkkin -min 0.1 in
Doing command: set_output_delay -clock clkoutt -max 9.9 dout
Error: Cannot find clock 'clkoutt'. (UID-250)
Doing command: set_output_delay -clock clkout -min -1.0 dout
```

Much better. I can see each command as it is executed, so I can tell exactly where the error occurred. &cmd has the additional benefit of reducing all my nested calls and expr's to their final result so that I can see EXACTLY what PrimeTime/DC executed.

The problem is that collections just echo out as “\_sel<some\_number>”. This isn't very helpful. We need to expand that collection into something meaningful.

### 3.3 &list\_coll

To do this expansion, we'll write a proc called “&list\_coll”. This proc takes a collection as its argument, and outputs a list. Isn't this exactly what query\_objects does? Well, yes and no. Query\_objects, like several of Synopsys commands, sends its output only to STDERR, not STDOUT (or whatever these concepts are called in Tcl). This is probably best illustrated with an example:

```
dc_shell-t> query_objects [get_cells *]
{"doutpad", "clktree", "clkoutpad", "dout_reg"}
dc_shell-t> set _cells [query_objects [get_cells *]]
{"doutpad", "clktree", "clkoutpad", "dout_reg"}
Information: Defining new variable '_cells'. (CMD-041)
dc_shell-t> echo $_cells

dc_shell-t>
```

As you can see, “query\_objects” by itself echos the contents of the collection. But it doesn’t return this information to the Tcl equivalent of STDIN for use by other Tcl commands. So, you can’t use it as an input to a set command, as illustrated above.

Anyway, &list\_coll is a close cousin of the &dump\_coll proc we saw earlier:

```
proc &list_coll args {
    # Call the standard parser
    &parse_proc_arguments -args $args results

    # Body of process
    if {$_limit < 0} {
        set _all 1
    } else {
        set _all 0
    }
    set _count 0
    set _list {}
    foreach_in_collection _item $_collection {
        if {!(($_all) && $_count >= $_limit)} {
            set _list [lappend _list "..."]
            break;
        }
        set _list [lappend _list [get_object_name $_item]]
        set _count [expr $_count + 1];
    }
    return $_list
}
&define_proc_attributes &list_coll -info "Create a list from a collection" \
    -define_args \
    {
        {-limit "Max to list - negative number means all" limit int optional 10}
        {_collection "Collection" target_collection list required}
    }
}
```

This is very similar to &dump\_coll. Note that I have added a feature to output all values, without limit, if the limit value is negative.

&list\_coll is required in &cmd to deal with collections. Many commands take collections as arguments, and it isn’t very helpful to see “Doing command: create\_clock -period 10.0 \_sel31”. The “\_sel31” is what PT/DCTL returns for a collection in a “string” context. It doesn’t help the debugging at all. We want to see the contents of the collection. That’s where &list\_coll comes in. But beware! You don’t want to go haphazardly changing collections into lists all over the place. Collections are used for a reason – they reduce memory usage and improve performance, so don’t just convert all collections to lists as a matter of course. &cmd does this for a limited number of items to make the “Doing command” output more meaningful.

## 3.4 &cmd

So, without further ado, here's &cmd:

```
if ![info exists _command_listmax] {set _command_listmax 5}
proc &cmd { args } {

    global _command_listmax

    # get rid of extra level of list structure in args
    # set args [eval concat $args]

    # get rid of any embedded cr's in args
    regsub -all {\n} $args {} args

    # echo result
    set _newargs $args
    foreach _arg $args {
        if {[regexp {_sel[0-9]+} $_arg _junk]} {
            regsub $_arg $_newargs "\{ [&list_coll -limit $_command_listmax $_arg]
\}" _newargs
        }
    }
    echo "Doing command: $_newargs"

    # Do command
    uplevel 1 $args ;# changed 1/9/2004
}
```

The global variable “\_command\_listmax” controls the `-limit` argument for `&list_coll`. I have defaulted it here to 5.

Let's go through &cmd step by step.

Notice the **commented-out** “eval concat \$args” line. This crushes out any list structure in the command. The reasons for this are lost in the mists of time. It had been that way in my code forever. One side effect of this step is to force curly braces to be escaped with a “\” in any situation involving explicit lists (like the `-waveform` argument of `create_clocks`). In the process of writing this paper, I began to question whether this step was really necessary. In the end, I concluded that it is not and I have since removed this (and all the escaped curly braces) from my core set of procs, and it all works fine. I left it in here as a comment because the new version doesn't have a lot of “runtime” on it.

The “regsub...” line eliminates any embedded newlines in the command. This allows &cmd to work on multi-line commands. The original code (before &cmd was inserted) escaped the newlines with “\”, but they will come back when &cmd does its eval if this step isn't done.

The next bit is how we get the collections turned into lists with `&list_coll`. The code is a little tricky. First we duplicate args into `_newargs`. Then we foreach our way through the elements in args, looking for entries that start with “\_sel” followed by a number. If we find such an entry, we substitute the value returned by `&list_coll` executed on this argument for the argument itself. In

other words, we substitute “{ foo bar }” for `_sel32`, where `_sel32` was a collection containing `foo` and `bar`. The global variable `_command_maxlist` is passed to `&list_coll` as the limit value so that we can control how long these lists get. Then, we echo `$_newargs` (which has all the substitutions) with the prefix of “Doing command: “.

Finally, we “uplevel 1” (equivalent of “eval”) the command, causing it to be executed by PT/DCTL.

As we saw before, here is what it looks like in action:

```
dc_shell-t> source -echo cmd_example_wcmd.dctl
set _someflag 1
if {$_someflag} {

    &cmd create_clock -period 10.0 [get_ports clkin]

    &cmd create_generated_clock      -name clkout      -source [get_ports
clkin]      -divide_by 1      [get_ports clkout]

    # create input constraint
    &cmd set_input_delay -clock clkin -max 9.95 in
    &cmd set_input_delay -clock clkin -min 0.1 in

    # create output constraint
    &cmd set_output_delay -clock clkoutt -max 9.9 dout
    &cmd set_output_delay -clock clkout -min "-1.0" dout

}
Doing command: create_clock -period 10.0 { clkin }
Doing command: create_generated_clock -name clkout -source { clkin } -
divide_by 1 { clkout }
Doing command: set_input_delay -clock clkin -max 9.95 in
Doing command: set_input_delay -clock clkin -min 0.1 in
Doing command: set_output_delay -clock clkoutt -max 9.9 dout
Error: Cannot find clock 'clkoutt'. (UID-250)
Doing command: set_output_delay -clock clkout -min -1.0 dout
1
```

Although I typically use `&cmd` with “real” commands, I believe it works with just about anything except “set”. The “set” command runs into trouble if it involves collections. My standard collection of procs is fully instrumented with `&cmd`, and I use it a lot in individual scripts as well.

## **4 Creating notifier force files from Primetime**

### **4.1 The problem – X’s from sync’ing flops create chaos in simulation**

In real-world designs, it is sometimes necessary to send signals from one clock domain to another. When this is done, the signal is generally run through a “synchronizer”, which usually consists of a synchronizing flop clocked in the receiving clock domain, followed by zero or more additional flops also clocked in the receiving domain.

Synchronizing flops are easy to handle in STA. I generally do `set_false_path` between ALL asynchronous domains (see Reference 1). Because by definition these paths go between asynchronous domains, they will never be seen. Even if you don’t routinely disable cross-clock paths, you can still safely `set_false_path` to them as you find them.

However, these flops create a problem in gate simulations, because they generally use regular flipflops, which will go “X” if setup or hold is violated. This X then propagates all through the simulation, and chaos ensues.

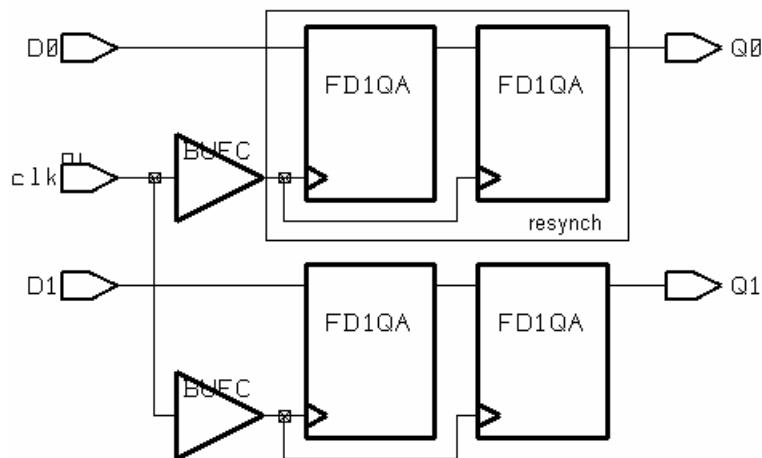
One way around this is to manipulate (perl to the rescue!) the netlist to change all the synchronizing flop instances to use some user-written model that won’t go X. This works, but there is a better way.

### **4.2 The solution – force the notifiers**

Most flipflop models in Verilog (sorry, VHDL folks, I’m a Verilog guy) do the X assertion using a device called a “NOTIFIER”. The setup/hold violation causes the NOTIFIER to toggle, which triggers the X behaviour. If you do a “force” on the NOTIFIER, the flop never goes X! And since “force” is an instance-specific command, you can do this on the synchronizing flop instances without messing up the X behaviour of the other flops.

This notifier force file can be written and maintained independently of STA, but if the designers use (as they should) as special design module for the resynchronizer, often the handiest way to generate the file is from PrimeTime.

Here's a sample circuit:



One designer has followed recommended practice and sync'ed D0 by instantiating an instance of the design "resynch". The other designer has been a bad boy (or girl) and just sync'ed D1 using ordinary rtl code.

What we want to end up with is a notifier force verilog file that looks something like this:

```
module force_notifiers();
  initial begin
    $display("Forcing notifiers on synchronizer flops at time %t", $time);
    force `TOP_PATH.resynchi.shifter_reg[0].notifier1 = 0;
    force `TOP_PATH.resynchi.shifter_reg[0].notifier2 = 0;
    force `TOP_PATH.rtl_reg[0].notifier1 = 0;
    force `TOP_PATH.rtl_reg[0].notifier2 = 0;
  end
endmodule
```

Notice the use of `TOP\_PATH to allow external code to `define the path into the chip. This path could also be the actual fixed path in simulation (and the proc's will allow that), but this approach is more flexible because the module can be used at multiple levels without regenerating the code.

### 4.3 Creating the notifier force file from PrimeTime

For reasons that will be clear shortly, I have chosen to break the creation of this file into three separate operations:

- 1) Create the module, initial, begin, and \$display statements
- 2) Create the actual force statements
- 3) Create the end, endmodule statements

To avoid maintaining extra procs, steps 1) and 3) share the same proc, with 3) using the -close option.



## 4.4 &create\_force\_file

The proc that creates the header and trailer code is called &create\_force\_file:

```
proc &create_force_file args {

    # Default optional arguments
    set results(-indent) " "
    set results(-close) 0
    set results(-append) 0
    set results(_display) "\${display}(\\"Forcing notifiers on synchronizer flops
at time \%t\", \${time})"
    set results(-top_path) "`TOP_PATH"
    set results(-outfile) "force_notifiers.v"
    set results(-module) "force_notifiers"

    # Call the standard parser
    parse_proc_arguments -args $args results

    &fixargs

    # Body of process
    if {$_append} {
        echo "Skipping file creation - append mode set"
    } elseif {$_close} {
    } else {
        echo "Creating nox file $_outfile"
        sh touch $_outfile
        sh rm -f $_outfile
    }

    # create file header
    if {$_close} {
        echo "${_indent}end" >> $_outfile
        echo "endmodule" >> $_outfile
    } else {
        echo "module ${_module}();" >> $_outfile
        echo "${_indent}initial begin" >> $_outfile
        if {[string length $_display] != 0} {
            echo "${_indent}${_indent}${_display};" >> $_outfile
        }
    }
}

define_proc_attributes &create_force_file -info "Create the notifier force
statement for a collection of registers" \
    -define_args \
    {
        {-indent "indent for formatting" indent string optional}
        {-close "Create closing code (default is opening)" "" boolean optional}
        {-append "Append to file instead of creating it" "" boolean optional}
        {-display "display statement" display string optional}
        {-top_path "Top level path" top_path string optional}
        {-outfile "Output file name" outfile string optional}
        {-module "module name" module string optional}
        {-verbose "Be Chatty" "" boolean optional}
    }
}
```

I have used **some** of the techniques the developed early to add options, etc. Since this code pre-dates `&define_proc_attributes` and `&parse_proc_arguments`, it doesn't use the fancy defaulting stuff.

There are a lot of options, but the code itself is quite simple. If the `--append` option wasn't selected, then I create the file by doing a `touch` (to make sure it exists) and then an `rm` via the `sh` (shell) command. I then switch on the `--close` option to either output the header or footer code. Most of the rest of the options relate to formatting and naming.

Why did I use `>>` redirection instead of `open/close/puts`? The truth is, I wrote this code a long time ago and I wasn't all that familiar with file management in Tcl. But this approach does have the advantage of removing the need to pass the file identifier around between the procs. And it works just as well.

The `--append` option is there to allow the calling code to create the file and put code into it before `&create_force_file` runs. I use this to default `TOP_PATH`:

```
echo "`ifdef TOP_PATH" > notifier_force.v
echo "`else" >> notifier_force.v
echo "  `define TOP_PATH top.bpia2" >> notifier_force.v
echo "`endif" >> notifier_force.v
```

Which puts this at the beginning of the `force_notifiers.v` file.

```
`ifdef TOP_PATH
`else
  `define TOP_PATH top.bpia2
`endif
```

Adding the `--append` option on the call to `&create_force_file` prevents this from being overwritten. This technique can also be used to add a `timescale` directive.

Anyway, when I run this:

```
pt_shell> &create_force_file
Creating nox file force_notifiers.v
```

I get:

```
module force_notifiers();
  initial begin
    $display("Forcing notifiers on synchronizer flops at time %t", $time);
```

So far, so good. Now to create the actual force statements.

## 4.5 `&create_force_on_collection`

Take a close look at the force statements. Notice that there is not one force statement per flop, but two. This is because some libraries (notably lsi's) have flop models with multiple notifiers, all of which have to be forced. And some flops have more notifiers than others. Because of this, and because libraries might call their single notifier node "notifier", or "NOTIFIER", or "Notifier", we need so somehow KNOW what the mapping is from a given flop to its list of notifiers. Since the notifier node only exists in the verilog model, not in the db model, it must be brought in from the outside. I use an associative array called "\_refname2notifier" to hold this information.

This simple example has only a single type of flop, so the mapping looks like this:

```
set _refname2notifiers(FD1QA) {notifier1 notifier2}
```

Real implementations will need a whole sequence of these set commands somewhere in the code. But don't despair – perl to the rescue! The appendix contains a perl script that will create this code by running over the flop verilog models.

Back to our example. The good designer has used our standard resynch module, so finding his resynch'ing flops is automatic:

```
set _resynch_instances \  
  [get_cells "*" -quiet -hier -filter "ref_name =~ resynch"]
```

The flops are [get\_object\_name \_resynch\_instance]/shifter\_reg[0].

The bad designer has used rtl. This means much hunting around, gnashing of teeth, emails back and forth, etc, but eventually the poor STA engineer determines that rtl\_reg[0] is a sync'ing flop.

```
set _sync_flops [get_cells rtl_reg[0] ]
```

So, we need a proc that knows (via global) \_refname2notifiers and creates force statements for each flop in our two collections.

There's a subtle issue here, though. The \_resynch\_instances is a collection of instances one level up from the actual flop, while \_sync\_flops is a collection of the actual flops. We could introduce code to add the /shifter\_reg[0] to \_resynch\_instances before calling the proc, but since this happens so often, I find it more convenient to make the proc handle both cases. This is done by adding an option called "-reg\_name".

This is one of the reasons I structure the functionality into separate proc's. That way I can call it for different sync'ing flop groups either with or without "-reg\_name".

Here's &create\_force\_on\_collection:

```
proc &create_force_on_collection args {

    global _refname2notifiers
    global _unmatched_refs

    # Default optional arguments
    set results(-indent) "    "
    set results(-release) 0
    set results(_reg_name) ""
    set results(-top_path) "`TOP_PATH"
    set results(-outfile) "force_notifiers.v"

    # Call the standard parser
    parse_proc_arguments -args $args results

    &fixargs

    # Body of process

    # This gets a little confusing. Verilog wants "." as the hierarchical
    # separator, but ptime wants "/". So, we create versions with both.
    if {[string length $_reg_name] != 0} {
        set _reg_name_wdots "${_reg_name}."
        set _reg_name_wslashes "/"${_reg_name}"
    } else {
        set _reg_name_wdots ""
        set _reg_name_wslashes ""
    }

    foreach_in_collection _instance $_collection {
        set _instance_name [get_object_name $_instance]
        # Find the ref_name (type) for use in foreach below
        set _refname [get_attribute \
            [get_cells ${_instance_name}${_reg_name_wslashes}] \
            ref_name \
        ]

        # Create the instance name with . instead of /
        regsub -all {/} $_instance_name {.} _instance_name_wdots

        if {[array names _refname2notifiers $_refname] != ""} {
            # Create one force for each notifier listed for this type of flop in
            # refnames2notifiers
            foreach _notifier $_refname2notifiers($_refname) {
                if {$_release} {
                    if {$_verbose} {echo "Creating release on
${_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier}"
                    echo "${_indent}release
${_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier};" >>
$_outfile
                } else {
                    if {$_verbose} {echo "Creating force on
${_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier}"
                    echo "${_indent}force
${_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier} = 0;" >>
$_outfile
                }
            }
        }
    }
}
```

```

    }
  } else {
    # Warn user if entry didn't exist
    echo -n "Err" ; echo "or: No entry in _refname2notifiers for $_refname
used by cell ${_instance_name}${_reg_name_wslashes}"
    set _unmatched_refs($_refname) 1
  }
}
}
define_proc_attributes &create_force_on_collection -info "Create the notifier
force statement for a collection of registers" \
  -define_args \
  {
    {_collection "Collection" target_collection list required}
    {-indent "indent for formatting" indent string optional}
    {-top_path "Top level path" top_path string optional}
    {-outfile "Output file name" outfile string optional}
    {-reg_name "Register name if collection isn't leaf cell" _reg_name string
optional}
    {-release "Create release statement" "" boolean optional}
    {-verbose "Be Chatty" "" boolean optional}
  }
}

```

Before going over the code in detail, there are a few options that need to be explained.

“-top\_path” allows you to change the path to the flops. It defaults to `TOP\_PATH.

“-outfile” allows you to change the output file name. It defaults to force\_notifiers.v

“-release” allows you to create release statements instead of force statements. Why would you want to do this? Well, there are some cases where you want certain flops to have their notifiers forced only for some short period at initialization. Having this option allows you to call the proc once to create the forces, then echo the “#time” statement, then call the proc again to create the releases. This is another reason why I structure the functionality in separate proc’s.

So, now let’s follow the code in a little more detail.

This part:

```

# This gets a little confusing. Verilog wants "." as the hierarchical
# separator, but ptime wants "/". So, we create versions with both.
if {[string length $_reg_name] != 0} {
  set _reg_name_wdots "${_reg_name}."
  set _reg_name_wslashes "/"${_reg_name}"
} else {
  set _reg_name_wdots ""
  set _reg_name_wslashes ""
}

```

just creates the variable `_reg_name`. If `-reg_name` was not set, `$_reg_name` will be set to blank. If `-reg_name` was set, we need to create `$_reg_name` in two forms – with “/” as a hierarchy divider for ptime accesses, and with “.” as a hierarchy divider for verilog. In the process, we

prepend the “/” for ptime and append a “.” for verilog. If you follow the code you’ll see that this works out nicely later on.

Now we start to foreach our way through the collection:

```
foreach_in_collection _instance $_collection {
```

First we get the name of the instance (rather than the pointer to it that’s in the foreach):

```
    set _instance_name [get_object_name $_instance]
```

Then we use get\_attribute to find the cell type (ref\_name):

```
    # Find the ref_name (type) for use in foreach below
    set _refname [get_attribute \
        [get_cells ${_instance_name}${_reg_name_wslashes}] \
        ref_name \
    ]
```

Now we create a verilog (“.”) version of the instance name:

```
    # Create the instance name with . instead of /
    regsub -all {/} $_instance_name {.} _instance_name_wdots
```

Next we need to find the list of notifiers for this cell type (ref\_name) in \_refname2notifier. If we don’t find it, we’ll issue a warning and add the instance to a list of cells that had this problem. This list is passed back by declaring it global at the beginning:

```
    if {[array names _refname2notifiers $_refname] != ""} {
        .
        .
        .
    } else {
        # Warn user if entry didn't exist
        echo -n "Err" ; echo "or: No entry in _refname2notifiers for $_refname
used by cell ${_instance_name}${_reg_name_wslashes}"
        set _unmatched_refs($_refname) 1
    }
```

If we do find it (the “..”), we do the actual code creation (based on release):

```
# Create one force for each notifier listed for this type of flop in
# refnames2notifiers
foreach _notifier $_refname2notifiers($_refname) {
  if {$_release} {
    if {$_verbose} {echo "Creating release on
$_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier}"}
    echo "${_indent}release
$_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier};" >>
$_outfile
  } else {
    if {$_verbose} {echo "Creating force on
$_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier}"}
    echo "${_indent}force
$_top_path}.${_instance_name_wdots}.${_reg_name_wdots}${_notifier} = 0;" >>
$_outfile
  }
}
```

(note that the embedded newlines are a formatting issue with Word – they aren’t in the real code).

That’s it.

We use the new procs on our example circuit like this:

```
set _refname2notifiers(FD1QA) {notifier1 notifier2}

# Create the header stuff
&create_force_file

# resynchs
set _resynch_instances \
[get_cells "*" -quiet -hier -filter "ref_name =~ resynch"]
&create_force_on_collection \
-reg_name shifter_reg[0] \
$_resynch_instances

# sync flops that are NOT in known sync flop modules
set _sync_flops [get_cells rtl_reg[0] ]
&create_force_on_collection $_sync_flops

&create_force_file -close
```

Which results in a force\_notifiers.v like this:

```
xd$ cat force_notifiers.v
module force_notifiers();
  initial begin
    $display("Forcing notifiers on synchronizer flops at time %t", $time);
    force `TOP_PATH.resynchi.shifter_reg[0].notifier1 = 0;
    force `TOP_PATH.resynchi.shifter_reg[0].notifier2 = 0;
    force `TOP_PATH.rtl_reg[0].notifier1 = 0;
    force `TOP_PATH.rtl_reg[0].notifier2 = 0;
  end
endmodule
```

It's just that easy!

## 5 More Fun Stuff

### 5.1 Defaulting variables using &default

Earlier, I defaulted a variable by doing this:

```
if ![info exists _command_listmax] {set _command_listmax 10}
```

But that gets tedious, since you have to remember to change the variable name in two places when you cut and paste it. Instead, let's put it in a proc!

```
proc &default args {
    # Call the standard parser
    &parse_proc_arguments -args $args results

    # Body of process
    set _cmd "info exists $_name"
    if ![uplevel 1 $_cmd] {
        set _cmd "set $_name \{$_value\}"
        if {$_verbose} {
            echo "-> Defaulting variable \"$_name\" to \"$_value\""
        }
        uplevel 1 $_cmd
    }
}
&define_proc_attributes &default -info "Default a variable in the current
scope" \
    -define_args \
    {
        {_name "Name of variable" name string required}
        {_value "Default value" value string required}
        {-verbose "Be Chatty" "" boolean optional}
    }
}
```

First, we create a variable `_cmd` and set it to “info exists `$_name`”. We then execute this in the calling scope by using “uplevel 1”. This will return true if the variable already exists. So, if it returns false (“if ![uplevel 1 `$_cmd`]”) then the variable does not exist and we should create it.

To create it, we put “set `$_name` \{`$_value`\}” into `_cmd` and use `uplevel` to execute the command in the calling scope. The escaped curly braces make the proc work even when value is the null string.

Here's what it looks like in action:

```
pt_shell> &default _foo 3
Information: Defining new variable '_foo'. (CMD-041)
3
pt_shell> echo $_foo
3
pt_shell> set _foo 4
4
pt_shell> &default _foo 3
pt_shell> echo $_foo
4
```

And it even works (thanks to the curly braces) when the value is "" or {}:

```
pt_shell> &default _bar ""
Information: Defining new variable '_bar'. (CMD-041)
pt_shell> echo $_bar

pt_shell> unset _bar
pt_shell> &default _bar {}
Information: Defining new variable '_bar'. (CMD-041)
pt_shell> echo $_bar

pt_shell>
```

## 5.2 Using `suppress_message` and `unsuppress_message` to control warning messages.

That warning in the example above:

```
Information: Defining new variable '_bar'. (CMD-041)
```

can get a little annoying. You can turn it off using “`suppress_message`”:<sup>4</sup>

```
pt_shell> suppress_message CMD-041
pt_shell> unset _bar
pt_shell> &default _bar {}
pt_shell> echo $_bar

pt_shell>
```

---

<sup>4</sup> In the case of CMD-041, you can also suppress this by setting the Synopsys built-in variable `sh_new_variable_message` to false.

And turn it back on with “`unsuppress_message`”:

```
pt_shell> unsuppress_message CMD-041
pt_shell> unset _bar
pt_shell> &default _bar {}
Information: Defining new variable '_bar'. (CMD-041)
pt_shell> echo $_bar

pt_shell>
```

The best part about `suppress_message` and `unsuppress_message` is that they work as a counter. Each call to `suppress_message` increments the counter (for this message) by one, and each `unsuppress_message` decrements by one. As long as the counter is non-zero, the message is suppressed:

```
pt_shell> suppress_message CMD-041
pt_shell> suppress_message CMD-041
pt_shell> unsuppress_message CMD-041
pt_shell> unset _bar
pt_shell> &default _bar {}
pt_shell> unsuppress_message CMD-041
pt_shell> unset _bar
pt_shell> &default _bar {}
Information: Defining new variable '_bar'. (CMD-041)
pt_shell>
```

In the example above, I made two calls to `suppress_message`, then one to `unsuppress_message`. Thus, the message is still suppressed. After I make a second call to `unsuppress_message`, the message is no longer suppressed.

This “counter” behaviour is really great. It means that procs and sourced files can mess around with message suppression all they like as long as they undo what they did, without having to know what other code is doing.

And, no, there isn’t a “-force” option on `unsuppress_message` to force the counter to zero.

## **6 Conclusion**

So, those are a few of my favorite DC/PT Tcl tricks. I hope you have found something in here that you can apply in your own work.

## **7 Acknowledgements**

The author would like to acknowledge the following people for their assistance and review:

Andy Siegel (Synopsys)  
Steve Brown (Corrent)  
Steve Golson (Trilobyte)

## **8 References**

- (1) Complex Clocking Situations Using PrimeTime  
Paul Zimmer  
Synopsys Users Group 2000 San Jose

## 9 Appendix

### 9.1 Perl script to create `_refnames2notifiers`

Here's a simple little perl script to create the `_refname2notifiers` array. It may not work on all libraries, but should be easy to tweak if it doesn't.

```
#!/usr/local/bin/perl
# mk_notifier_xref.pl - make the _refname2notifiers array for force_notifiers
# Usage:
#   cat *.v | mk_notifier_xref.pl
#   or
#   mk_notifier_xref.pl *.v

# troll through the code looking for module and notifier statements
while (<>) {
  if (/^\s*module\s+(\S+)\s*\s*(\/) {
    # matched module - save it
    $module = $1;
  } elsif (/^\s*reg.*notifier/) {
    # matched reg declaration for notifier
    # allow for multiple notifiers in one reg statement
    @notifiers = /(notifier[\d\w]*)/g;
    # %nots is a hash of arrays
    push @{$nots{$module}}, @notifiers;
  }
}

# print out contents of %nots
foreach $module (sort keys %nots) {
  $line = "    set _refname2notifiers({$module}) {";
  for $notifier (@{ $nots{$module} }) {
    $line .= "$notifier ";
  }
  $line =~ s/ $/}/;
  print "$line\n";
}
```