“Herding cats”
Keeping your sdc code sane in a multi-tool environment

Paul Zimmer
Zimmer Design Services

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

website: www.zimmerdesignservices.com

ABSTRACT

The task of creating and maintaining correct timing constraints (SDC) has become increasingly complex over the years. Not only is the timing environment (clocks, generated clocks, exceptions, etc) complex, but it has to work at multiple levels of hierarchy and in multiple places in the flow that use diverse tools - synthesis, place and route, and signoff STA.

This can make it difficult to have a single, uniform sdc script. The result is often a mishmash of bits and pieces of scripts cut and pasted together for each particular phase and tool.

This paper will describe the author's techniques to rein all of this in and keep to a single source for all sdc.

Table of contents

1 Introduction - It's all just sdc, right?

The task of creating and maintaining correct timing constraints (SDC) has become increasingly complex over the years. Not only is the timing environment (clocks, generated clocks, exceptions, etc) complex, but it has to work at multiple levels of hierarchy and in multiple places in the flow that use diverse tools - synthesis, place and route, and signoff STA.

While sdc is technically "standardized", it can be very time consuming and error-prone to write and maintain the code using only the minimum subset shared by all tools ("baby sdc"). Access to tool-dependent features (attributes, find/filter, etc) is often essential in writing clear and maintainable code.

In addition, some values and techniques may change according to the tool and the phase in the flow. I/O constraints are often simplified for synthesis and P&R. Clock uncertainties are different. Some exceptions might be applied at particular steps in the flow to produce the desired results. And on and on.

But, despite all the differences, the fundamental core of the sdc will likely remain unchanged.

The result is often a mish-mash of chunks of code being copied and pasted between different sdc files at random times for the different tools and phases - a very error-prone process.

What is needed is a single, common sdc script that can be used in all tools and in all phases but still allows for the differences.

This is my attempt to create such a flow.

2 Portable procs

One thing that would help would be a way to abstract the higher-level operations in such a way that the tools-specific operations can be hidden and re-used, and that allow for different values to be applied depending on tool and phase.

The obvious solution is to wrap these higher level operations in procs (subroutines). Then, you can either load a different set of procs depending on the tool and/or phase, or bury the complicated "if {$tool eq "primetime"}" stuff inside the proc. Or, you can use a mix of these strategies.

For example, PT likes time units to be the same as in the primary library - usually nanoseconds. But RC really wants picoseconds. So, apply constants using a proc. The proc "&ps" takes a value that is assumed to be in picoseconds and returns the value in whatever is appropriate for that tool. In RC, it would just return what it was given. In PT, it would return the value given divided by 1000.

So, &ps is an example of a proc that depends on the tool. &setup_uncertainty is a proc that depends on the phase. Given a clock period, it will return the correct uncertainty (using &ps to get the correct units) depending on global variable that gives it the phase of the flow (synthesis, STA).

Note I haven't mentioned P&R. While this technique could be extended to P&R, I generally find it handier to write the P&R sdc out of synthesis, after first removing things I don't want (like large uncertainties). But the technique would work equally well for P&R tools.

2.1 Other reasons to use procs

Besides making it easier to make the same set of constraints work in different tools and phases, there are other excellent reasons to use procs.

2.1.1 Manage hierarchy using procs

If you have a generated clock inside some low-level subblock, that create_generate_clock command is going to have to be replicated with a slightly different path many times in the higher levels in the hierarchy. This is tedious and error-prone.

You could "source" a file with this command, and pass the path via a variable, but that results in many small files to manage, and/or complicated "if" flows in the block level sdc.

Instead, wrap the create_generated_clock in a proc, with a "-path" option. This proc is then called in the local block synthesis, then called by the next level up via its own "create_generated_clock" proc, which also has a -path option. And so on up the stack. This will be discussed in greater detail later.

2.1.2 Keep related code together, even when not executed sequentially

Constraining i/o's often requires several steps - creating a clock or generated clock, applying input/output delays to ports, running i/o specific reports, etc. Often, these steps are done in different parts of the flow - clock creation with the other clock creation, reports with reports, etc.

By wrapping these operations in procs, it is possible to keep them all in a single file and still call them at any time in any order. Such a structure also makes it much easier to re-use the code. You pass it port names as arguments, for example. Future chips can re-use the code and just pass a different port list.

2.1.3 Simplifies debug

The ability to call up large pieces of code without cutting/pasting/tempfiles is very valuable during interactive debug.

2.1.4 Structured code is just plain better

Early in the history of computing it was discovered that subroutines make code less buggy, easier to read, and easier to maintain than inline code.

SDC is no exception!

2.2 Why people don't use procs

So, why don't people use procs? In a word, because it's *hard*, primarily because option parsing is a mess. The sdc tools all use tcl, but they don't have a standardized way of parsing options - there's no "GetOpts" in tcl.

All the tools have GetOpts-like capabilities, but *they're all different!*

What this means is that any proc, even if it doesn't use any tool-specific features, will have to have its option parsing re-written and the proc will have to be separately maintained - forever. This is a big disincentive to using procs.

You could avoid this by doing all the option parsing yourself, directly in tcl. But that is some very messy code - an even bigger disincentive to using procs.

2.3 Tool-independent option parsing - &define_proc_attributes and &parse_proc_arguments

What we need is a tool-independent way to parse proc arguments. I have developed such a technique.

Most of my sdc code base, including procs, was developed in a design_compiler + primetime environment. These tools have two "helper" procs that do the equivalent of GetOpts:

- define_proc_attributes - Defines types (boolean, string, list), optionality (required/optional), help information, etc
- parse_proc_arguments - actually does the parsing based on the information provided by define_proc_attributes when the proc is called.

Back in 2003, I presented a paper at the Synopsys Users Group (SNUG) conference entitled "My Favorite DC/PT Tcl Tricks" that described how to put wrappers around these procs to make them even more powerful (provide defaults, for example).

Since my flow is based on procs, when I started with RC, the first thing I did was go looking for define_proc_attributes and parse_proc_arguments. No luck. So, I learned the RC equivalent - parse_options. Then I started the laborious process of converting all my handy procs for use in RC.

Somewhere along the line, it occurred to me that I could use my wrappers for the DC/PT option parsing procs to generate parse_options code. The new procs, &define_proc_attributes and &parse_proc_arguments, give me a tool-independent way to parse proc options.

2.3.1 What they do

My proc option parsing commands &define_proc_attributes and &parse_proc_arguments combine to do several things:

1. Provide tool-independent parsing of proc options.
2. Provide a simple way to define the default value of any variable in the argument declaration (an extra value at the end).
3. For any "-" argument that has a defined value, create a variable of that name prefixed by "_" containing the value (ex: "-foo bar" creates a variable _foo with the value bar)
4. Create all boolean arguments according to the rule above, and set their value to 0 unless a different default value is given.
5. Allow "-no" use on a boolean (ex: -noverbose force _verbose to 0 regardless of default).

2.3.2 An illustrative example

Without going into the implementation details, here's an example:

```
proc &proc_template { args } {

  # Call the parser
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  # This makes it easier to debug by globalizing some internal variables
  if {$_debug} {
    global internal_var
  }

  puts "-verbose : \"$_verbose\""
  puts "-debug : \"$_debug\""
  if {[info exists _float_option]} {puts "-float_option : \"$_float_option\""}
  if {[info exists _string_option]} {puts "-string_option :
\"$_string_option\""}
  puts "-float_required : \"$_float_required\""
  puts "-list_required : \"$_list_required\""
  puts "-list_opt_with_default : \"$_list_opt_with_default\""
  puts "_string_arg : \"$_string_arg\""

}
&define_proc_attributes &proc_template -info "" \
  -define_args \
  {
    {-verbose "Be Chatty" "" boolean optional}
    {-debug "Debug flow" "" boolean optional 1}
    {-float_option "Numeric option - optional" float_option float optional}
    {-string_option "String option - optional" string_option string optional}
    {-float_required "Numeric option - required" float_required float
required}
    {-list_required "List option - required" list_required list required}
    {-list_opt_with_default "List option - optional" list_opt_with_default
list optional [list]}
    {_string_arg "Positional string argument" _string_arg string required}
  }
```

Let's look at the &define_proc_attributes part first.

Since this started as a wrapper for the define_proc_attributes command of DC/PT, the fields are taken from that command (with the exception of the last field, which is my own invention):

- The first field is is the option itself. An option that starts with "-" means that the option is not postitional; the option name is required in the invocation "ex: -foo bar". An option that starts with "_" is a positional option. "ex: bar".

- 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.

- 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. Don't use this as it isn't supported by my published procs.

- The sixth field is the default value, if any (this field can be omitted).

So, the first line of our example defines an option "-verbose" which is a boolean. The second defines an option "-debug" that is a boolean, but defaults to "1". The third line defines an option that is required to be numeric (a floating point number), but is optional. The fourth lines does the same thing for a string option.

The fifth line defines an "option" (meaning it uses -float_required) that is required. This might sound odd, but it is actually quite useful. It allows for required arguements, but avoids using positional arguments, which can be error-prone.

The sixth line defines a required list option, the seventh an optional list *with a default*, and the last line a positional argument of type string.

Note that &define_proc_attribute is a itself a proc. So, when the code above gets read into the tool, it will be executed. &parse_proc_arguments doesn't get executed until it is called by the proc &proc_template when &proc_template is invoked.

Now let's look at the body of the proc. The first thing is a call to &parse_proc_arguments that looks like this:

```
  # Call the parser
  if {[&parse_proc_arguments -args $args results] != 1} { return }
```

Why so complicated? Frankly, I forget. If you just call &parse_proc_arguments directly, there is some corner case that creates bogus errors. Long ago, I figured out that this code avoids the problem. So, just do it this way.

I also threw this little tidbit into the body of the proc:

```
  # This makes it easier to debug by globalizing some internal variables
  if {$_debug} {
    global internal_var
  }
```

I often use this construct in my procs. Basically, I globalize a lot of my internal variables if debug mode is set. This makes it easier to see what happened after the proc has run.

Note that I can just refer to the variable $_debug without any setup or anything. I defined a boolean "-debug" in &define_proc_attributes, so &parse_proc_arguments created a variable $_debug for me. I specified the default value as "1", so unless the invocation had "-nodebug", the $_debug variabl will have the value "1".

The rest of the proc just prints the values:

```
  puts "-verbose : \"$_verbose\""
  puts "-debug : \"$_debug\""
  if {[info exists _float_option]} {puts "-float_option : \"$_float_option\""}
  if {[info exists _string_option]} {puts "-string_option :
\"$_string_option\""}
  puts "-float_required : \"$_float_required\""
  puts "-list_required : \"$_list_required\""
  puts "-list_opt_with_default : \"$_list_opt_with_default\""
  puts "_string_arg : \"$_string_arg\""
```

Note that for booleans and required arguments, and for optional arguments with defaults ("-list_opt_with_default), it can just print the argument. As explained above, the variable *will* exist once &parse_proc_arguments has run. For optional arguments without defaults, it checks first to make sure the variable exists. It will only exist if that option was invoked.

There is an option to &define_proc_attributes (-default_string_to_null) that will cause unused vars to get set, but this is deprecated because it can be difficult within the body of the proc to know whether an option variable was set to null by the invocation of the command or by the defaulting code. Leaving it unset is more useful.

So, lets take a look at &proc_template in action.

First, we'll invoke it with no options:

```
rc:/> &proc_template
Error   : A required argument was not specified. [TUI-202] [parse_options]
        : The flag '-float_required' was not specified.
        : Rerun the command specifying all required arguments.

Usage: &proc_template [-verbose] [-debug] [-nodebug] [-float_option <float>]
[-string_option <string>]
```

```
           -float_required <float> -list_required <string> [-
list_opt_with_default <string>] <string> [> file]

    [-verbose]:
        Be Chatty
    [-debug]:
        Debug flow
    [-nodebug]:
        Turn off Debug flow
    [-float_option <float>]:
        Numeric option - optional
    [-string_option <string>]:
        String option - optional
    -float_required <float>:
        Numeric option - required
    -list_required <string>:
        List option - required
    [-list_opt_with_default <string>]:
        List option - optional
    <string>:
        Positional string argument
```

As expected, we get an error message because we didn't specify the required options. So, we'll do that:

```
rc:/> &proc_template -float_req 10.0 -list_required [list foo bar]
positional_string
-verbose : "0"
-debug : "1"
-float_required : "10.0"
-list_required : "foo bar"
-list_opt_with_default : ""
_string_arg : "positional_string"
```

Note that (unlike in the native RC parser), we do not have to put the positional argument last:

```
rc:/> &proc_template positional_string -float_req 10.0 -list_required [list
foo bar]
-verbose : "0"
-debug : "1"
-float_required : "10.0"
-list_required : "foo bar"
-list_opt_with_default : ""
_string_arg : "positional_string"
```

Remember that -debug is defaulted to 1. We can override this by using -nodebug:

```
rc:/> &proc_template positional_string -float_req 10.0 -list_required [list
foo bar] -nodebug
-verbose : "0"
-debug : "0"
-float_required : "10.0"
-list_required : "foo bar"
-list_opt_with_default : ""
_string_arg : "positional_string"
```

2.3.3 A peek under the hood

I don't want to get too deep into the internals of these procs, but it might be useful to give a general overview of how they work.

In RC, most of the work is done by &define_proc_attributes. What it does is to parse the arguments given to it, then create an entry for that proc in a global array I call "_parse_options" that contains the "parse_options" code that should be run when the proc is invoked:

```
rc:/> echo $_parse_options(&proc_template)
switch -- [ \
    parse_options &proc_template file_var $args \
      {-verbose bos Be Chatty} _verbose \
      {-debug bos Debug flow} _debug \
      {-nodebug bos Turn off Debug flow} _nodebug \
      {-float_option fos Numeric option - optional} _float_option \
      {-string_option sos String option - optional} _string_option \
      {-float_required frs Numeric option - required} _float_required \
      {-list_required srs List option - required} _list_required \
      {-list_opt_with_default sos List option - optional}
_list_opt_with_default \
      {srs Positional string argument} _string_arg \
    ] {
      -2 {return}
      0 {return -code error}
    }
```

To support the default capability, &define_proc_attributes creates an entry in another array I call "_parse_defaults" that handles the defaulting:

```
rc:/> echo $_parse_defaults(&proc_template)

if {![string compare $_verbose ""]} {set _verbose 0}
if {![string compare $_debug ""]} {set _debug "1"}
set _debug 1
if {$_nodebug == 1} {set _debug 0}
if {[string compare $_float_option ""]} {set _float_option [expr
double($_float_option)]}
if {![string compare $_float_option ""]} {unset _float_option}
if {![string compare $_string_option ""]} {unset _string_option}
if {[string compare $_float_required ""]} {set _float_required [expr
double($_float_required)]}
if {![string compare $_list_opt_with_default ""]} {set _list_opt_with_default
"[list]"}
```

In addition to setting the defaults, this code will undo the defaults normally set by "parse_options".

Also, parse_options has the peculiar behavior of returning arguments specified as type "float" as a string - a string that converts to an integer:

```
proc &parse_options_test args {
```

```
 switch -- [ \
    parse_options &proc_template file_var $args \
      {-float_required frs Numeric option - required} _float_required \
    ] {
      -2 {return}
      0 {return -code error}
    }

  puts "_float_required: $_float_required"

}
```

```
rc:/> &parse_options_test -float_required 10.0
_float_required: 10
```

That's why &parse_proc_attributes inserts code to turn floats returned by parse_options into true floats (the [expr double...] part).

So, when the proc is invoked, all &parse_proc_arguments has to do is eval these array entries:

```
  proc &parse_proc_arguments { args } {
    set calling_proc [uplevel 1 &myname]

    uplevel 1 $::_parse_options($calling_proc)
    uplevel 1 $::_parse_defaults($calling_proc)

    # This allows the calling proc to determine whether this proc completed.
    return 1
  }
```

Note that it does the eval using "uplevel 1" to make the commands take effect in the calling proc's namespace.

That's it.

3 Handling missing commands

3.1 RC commands that are missing in PT

There are a few commands that are essential in RC but have no equivalent in PT. The easy way around this to create procs with that command name and either implement the functionality or just return.

One example is vname. In RC, vname converts a path (/design/../subdesign/..) into a verilog path. This is unnecessary in PT, but I don't want to put "if {$::tool eq "rtl_compiler"}" code around every use of vname. Instead, I'll just create the vname proc in PT and have it return its argument.

Better yet, since the argument might be a collection, have it return the list in place of the collection.

```
proc vname args {
  # vname in RC returns the verilog name.  The same thing here, except that we
might
  # have to convert from a collection to a list
  if {[regexp {^_sel\d+} $args]} {
    return [get_object_name $args]
  } else {
    return $args
  }
}
```

Other RC commands that get this treatment are set_time_unit (do nothing), dirname (get the stuff up to the last /), and basename (get the stuff after the last /).

```
proc set_time_unit args {
}

proc dirname args {
  return [regsub {/[^\/]+$} $args {}]
}

proc basename args {
  return [regsub {^.*/} $args {}]
}
```

3.2 PT commands that are missing in RC

Interestingly, I haven't really found any commands in PT that don't exist in RC. The problem is that the PT commands in RC belong to a tcl namespace called "dc". When you're running the "read_sdc" command, this namespace is somehow magically invoked, so you don't have to preface your commands with "dc::" or "::dc::".

But if you need to use PT commands outside of read_sdc (which I do sometimes), you have to use the "dc::" prefix (example: dc::sizeof_collection).

I get around this by duplicating the namespace in PT, remapping all dc::xyz commands to xyz. Then I just always use the dc:: prefix whenever I need to use a PT command in code that will execute in RC outside of the read_sdc environment.

The magic tcl code looks something like this:

```
# Set up dummy dc:: version of commands for compatibility with rc code
namespace eval dc {
  foreach cmd [list \
    add_to_collection \
    all_clocks \
    all_fanin \
    all_fanout \
    all_inputs \
    all_outputs \
  ] {
    eval "proc $cmd args \{ uplevel 1 $cmd \$args \}"
  }
}
```

Proof is left to the reader. :-)

The full code is in the appendix.

4 Managing Hierarchy

4.1 Shared code and unshared code

Within the sdc of a subblock, there are parts of the code that need to be re-used (shared) at higher levels in the compile hiearchy, and parts that do not.

The command "create_clock" is usually code that does not need to be re-used at higher levels. Higher level compiles will have already created a clock that filters down to the clock input of the subblock, so we don't want to create it again.

The command "create_generated_clock" is another matter. This usually *doe*s need to be re-used at higher levels, but the master, source of the master, and even the path to the source of the generated clock will be different. These differences are easily handled using procs as will be shown.

Exceptions, such as "set_multicycle_path" and "set_false_path" also usually need to be re-used and will also benefit from being put in a proc.

I handle this by having a block-specific "<blockname>_cgc" proc for create_generated_clock commands, and a block-specific <blockname>_exceptions" proc for exception commands.

4.2 A feline example

This is most easily illustrated by example. Suppose I have a top level called "cat" that instantiates a subblock called "kitten".

```
module cat (
  input wire     in0,
  output wir     out0,
  input wire     in1,
  output wire    out1,
  output wire    outcomb,
  input wire     clkzero,
  input wire     clkone,
  input wire     reset_n
);

kitten kitten (
          .in0                          (!in0),
          .out0                         (out0),
          .out1                         (out1),
          .outcomb                      (outcomb),
          .in1                          (in1),
          .clk0                         (clkzero),
          .clk1                         (clkone),
          .reset_n                      (reset_n));

endmodule
```

Note the name change

Kitten has 2 clock inputs, clk0 and clk1. It also creates a div-2 generated clock from clk0.

We create a kitten_cgc proc that looks like this:

```
proc &kitten_cgc args {

  global c

  # Call the parser
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  puts "Starting [&myname]"

  # Create the div2 clock
  &eval create_generated_clock -divide_by 2 -name $_name -source [&source_of
$_master] -add -master $_master [get_pins ${_path}out0_reg/$c(qpin)]

  puts "End of [&myname]"

}
&define_proc_attributes &kitten_cgc -info "Create generated clocks for kitten"
\
  -define_args \
  {
    {-master "master clock" master string optional clk0}
    {-name "name for generated clock" name string optional out0clk}
    {-path "path to kitten" path string optional ""}
  }
```

echos command, then executes it

No "/"

Special proc

Default value

This uses the parse option handling discussed earlier. The proc has three options:

- -master The master clock. Defaults to "clk0".
- -name The name of the new clock. Defaults to "out2clk".
- -path The hierarchical path. Defaults to "".

&eval is a proc of mine that echos "Doing command:" followed by the command (after conversion of collections to lists in PT/DC), then executes the command. It is explained in reference (1) and is included in the appendix.

Note the use of "&source_of". This proc returns the source of a clock. This will be discussed later.

Note also that the clock is created on "[**get_pins** ${_path}out0_reg/$c(qpin)]". When path is "" (the default, this becomes "[**get_pins** out0_reg/$c(qpin)]". This is a small, but important, detail. If I were to use "[**get_pins** ${_path}/out0_reg/$c(qpin)]" (with a "/"), then setting _path to "" wouldn't work. Thus, non-blank path values should always end in "/", and there should be no "/" after ${_path}.

kitten also has a timing exception - a multicycle path to an internal flop input pin. So, we create a kitten_exceptions proc like this:

```
proc &kitten_exceptions args {

  global c

  # Call the parser
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  puts "Starting [&myname]"

  &eval set_multicycle_path -setup 2 -to [get_pins ${_path}out0_reg/$c(dpin)]

  puts "End of [&myname]"

}
&define_proc_attributes &kitten_exceptions -info "Exceptions (false path,
multicycle path, etc) for kitten block" \
  -define_args \
  {
    {-path "path to kitten" path string optional ""}
  }
```

kitten.sdc (the subblock sdc) would then have this:

```
&eval create_clock -name clk0 -period $clk0per [get_ports clk0]
&eval create_clock -name clk1 -period $clk1per [get_ports clk1]
&kitten_cgc
&kitten_exceptions
```

There's no need for options on either proc, since the defaults work fine.

When executed, the output looks like this:

```
sdc_shell> &kitten_cgc
Starting &kitten_cgc
Doing command: create_generated_clock -divide_by 2 -name out0clk -source
/designs/kitten/ports_in/clk0 -add -master clk0
/designs/kitten/instances_seq/out0_reg/pins_out/q   ; Fri Jan 04 19:23:21 -
0800 2013 (abs: 1357356201 ) (time_since_last_cmd: 7 )
End of &kitten_cgc
sdc_shell> &kitten_exceptions
Starting &kitten_exceptions
Doing command: set_multicycle_path -setup 2 -to
/designs/kitten/instances_seq/out0_reg/pins_in/d   ; Fri Jan 04 19:23:31 -0800
2013 (abs: 1357356211 ) (time_since_last_cmd: 10 )
End of &kitten_exceptions
```

Now for cat.sdc.

The kitten's clk0 and clk1 pins are actually connected to the clkzero and clkone ports, and will have clocks named clkzero and clkone:

```
&eval create_clock -name clkzero -period $clkzeroper [get_ports clkzero]
&eval create_clock -name clkone -period $clkoneper [get_ports clkone]
```

These clocks go directly to kitten's clk0 and clk1 ports, so there is no need to create kitten's input clocks. But kitten also has a generated clock. &kitten_cgc will be called to create the generated clock. But the name of the master clock no longer matches the default (clk0). Also, we want to change the name of the generated clock to "clkzero_div2". Finally, the path to kitten is "kitten/", so we end up with this:

```
&kitten_cgc -path kitten/ -master clkzero -name clkzero_div2
```

When we run the command, we get this:

```
sdc_shell> &kitten_cgc -path kitten/ -master clkzero -name clkzero_div2
Starting &kitten_cgc
Doing command: create_generated_clock -divide_by 2 -name clkzero_div2 -source
/designs/cat/ports_in/clkzero -add -master clkzero
/designs/cat/instances_hier/kitten/instances_seq/out0_reg/pins_out/Q   ; Sun
Dec 30 14:28:07 -0800 2012 (abs: 1356906487 ) (time_since_last_cmd: 7 )
End of &kitten_cgc
```

Pretty straightforward. Now let's add a new wrinkle. Suppose cat has another mode that will be represented by creating a second clock on port clkzero called "clkthree":

```
&eval create_clock -name clkthree -period $clkthreeper [get_ports clkzero] -
add
```

But, since this goes into kitten on port clk0, it has a div2 generated clock as well. No problem, we just call kitten_cgc again:

```
&kitten_cgc -path kitten/ -master clkthree -name clkthree_div2
```

Because we were clever enough to write the proc using "-add" on create_generated_clock, this works fine:

```
sdc_shell> &kitten_cgc -path kitten/ -master clkthree -name clkthree_div2
Starting &kitten_cgc
Doing command: create_generated_clock -divide_by 2 -name clkthree_div2 -source
/designs/cat/ports_in/clkone -add -master clkthree
/designs/cat/instances_hier/kitten/instances_seq/out0_reg/pins_out/Q   ; Sun
Dec 30 14:31:18 -0800 2012 (abs: 1356906678 ) (time_since_last_cmd: 191 )
End of &kitten_cgc
```

Finally, we call kitten_exceptions with the path set to "kitten/":

```
&kitten_exceptions -path kitten/
```

Note that if we had two instantations of kitten, say kitten1 and kitten2, we would just call the &kitten_... procs twice with different paths.

That's all well and good if cat is the top level module, but suppose it is not?

In that case, cat would get its own "_cgc" and "_exceptions" procs. These then call the kitten versions, using "${_path}kitten/":

```
proc &cat_cgc args {

  global c

  # Call the parser
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  puts "Starting [&myname]"

  # Create the div2 clock inside kitten
  &eval &kitten_cgc -path ${_path}kitten/ -master $_master -name $_name


  puts "End of [&myname]"

}
&define_proc_attributes &cat_cgc -info "Create generated clocks for cat" \
  -define_args \
  {
    {-master "master clock" master string optional clkzero}
    {-name "name for generated clock" name string optional clkzero_div2}
    {-path "path to cat" path string optional ""}
  }

proc &cat_exceptions args {

  global c
```

```
  # Call the parser
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  puts "Starting [&myname]"

  # Call kitten exception
  &eval &kitten_exceptions -path ${_path}kitten/

  puts "End of [&myname]"

}
&define_proc_attributes &cat_exceptions -info "Exceptions (false path,
multicycle path, etc) for cat block" \
  -define_args \
  {
    {-path "path to cat" path string optional ""}
  }
```

We would then replace the "&kitten_" lines in the cat.sdc file with this:

```
&cat_cgc -master clkzero -name clkzero_div2
&cat_cgc -master clkthree -name clkthree_div2
&cat_exceptions
```

And higher level modules (what's higher than "cat"?) would call &cat_cgc, which would call &kitten_cgc. And so forth up the hierarchy.

The beauty of this is that the generated clock information *is in only one place*. If it changes (say, kitten is redesigned to use a modulated clock gator instead of a divider flop), the change is made in kitten_cgc and propagates everywhere.

5 Some very special procs - &create_clock and &create_generated_clock

5.1 There's more to creating clocks than just creating the clocks

Creating clocks and generated clocks have their own commands in sdc. While the commands themselves are standardized in sdc, there are many related operations that are not:

- Setting the uncertainties, which may be a function of the period or group of the clock (or period/group of the master clock in the case of a generated clock), as well as a function of the design phase (more uncertainty in synthesis than in STA).
- For a generated clock, finding (given the name of the master) the source of the master for the "-source" argument. This is better than hard-coding it so that if the source of the master changes you don't have to remember to change all the generated clocks.
- Putting the clock in the correct group (usually the same group as the master for a generated clock)

5.2 A related problem - we often need master, period, waveform, uncertainty, etc of clocks but the tools supply this information differently (or don't supply it at all)

When writing procs (or other sdc code) to perform common functions, you often want to know the master of a generated clock, or the period or uncertainty of a clock, and so forth. This information is sometimes available from the tool (period, for example), but the technique for extracting it is tool-dependent. Worse yet, some of this information, such as the master of a generated clock, isn't directly available in some tools at all and can only be extracted by the rather complex and runtime-intensive method of parsing reports.

5.3 The solution - wrap your clock creation commands and squirrel away the information in global arrays

After years of struggling with this, I finally decided to surrender to the inevitable and put wrappers around create_clock and create_generated_clock. This provides two advantages over the base sdc command. First, I can add options of my own to handle the related tasks (grouping, uncertainty, etc). Second, I can squirrel away all the clock data into global arrays as the clocks are created, making it easy to fetch, say, the master of a generated clock, later on.

5.4 &create_clocks

The wrapper around create_clock has two uses:

1. We can grab values, such as the period and the source, as the clock is created rather than have to query them somehow from the tool.
2. We can add new options to do things like setting the uncertainty or storing clock group information.

&create_clock has all the options of create_clock, plus a few of its own:

```
&define_proc_attributes &create_clock \
  -info "Wrapper for create_clock"  \
  -define args {
  {-add "Same as create_clock" "" boolean optional}
  {-name "Same as create_clock" name string optional}
  {-period "Same as create_clock" period float optional}
  {-waveform "Same as create_clock" waveform list optional}
  {-setup_uncertainty "Setup uncertainty" setup_uncertainty float optional}
  {-hold_uncertainty "Hold uncertainty" hold_uncertainty float optional}
  {-default_uncertainty "Use &setup_uncertainty and &hold_uncertainty for
uncertainty" "" boolean optional}
  {-group "Group in clkgroups" group string optional}
  {_src "Same as create_clock" src string optional ""}
}
```

The ones in ***bold/italics*** are unique to &create_clock.

We're going to use the following global arrays, indexed by the clock's name, to store information about the clock for later use:

- periodof
- waveformof
- sourceof
- setup_uncertaintyof
- hold_uncertaintyof

So, the first thing we do is globalize them for easier use from inside the proc:

```
proc &create_clock { args } {
  global c
  global periodof
  global waveformof
  global sourceof
  global setup_uncertaintyof
  global hold_uncertaintyof
```

The implementation of "-group" will be flow-dependent. Personally, I use an an array, indexed by group name, to keep track of the clock group information (more on this later). So, that gets globalized as well:

```
  global clkgroups
```

Next, we call &parse_proc_arguments as usual:

```
  if {[&parse_proc_arguments -args $args results] != 1} { return }
```

Tcl's "expr" command has some rather unexpected behavior. When given floating point values, it behaves as expected:

```
rc:/> expr 2.0 / 3.0
0.6666666666666666
```

But give it integers, and strange things happen:

```
rc:/> expr 2 / 3
0
```

For this reason, I always force the clock period to be a floating point number:

```
  # Make sure period is float
  set _period [expr double($_period)]
```

Now we can save the period in periodof:

```
  set periodof($_name) $_period
```

If "-waveform" was specified, we can insert this into waveformof. If not, assume 50% since that's what the tools do:

```
  if {[info exists _waveform]} {
    set waveformof($_name) $_waveform
  } else {
    set waveformof($_name) [list 0 [expr $_period/2]]
  }
```

Source is whatever is returned by "get_object_name". RC has implemented get_object_name for us. It probably just returns whatever it is sent.

```
  set sourceof($_name) [get_object_name $_src]
```

Now it's time to actually call create_clock. But with what arguments? We need to pass any arguments given on the command line. There are two ways to do this. We build up the arguments based on knowledge of the create_clock arguments, doing something like:

```
set cmd "create_clock"
if {[info exists _waveform]} {
  set cmd "$cmd -waveform $_waveform"
}
```

Or, we can use the $args passed to &create_clock and remove options specific to &create_clock.

Either way will work. I chose to remove the &create_clock arguments from $args:

```
  # Handle my args
  foreach {one_arg_opt_pattern} [list {-setup\S*} {-hold\S*} {-group\S*}] {
    regsub -- "${one_arg_opt_pattern}\\s+\\S+" $args {} args
  }
  foreach {boolean_opt_pattern} [list {-default\S*}] {
    regsub -- "${boolean_opt_pattern}\\s*" $args {} args
  }
```

Now, execute create_clock, up one level:

```
  &eval uplevel 1 "create_clock $args"
```

OK, time to start working on &create_clock's own arguments.

First, -group. The implementation of this will likely be different for different folks. As mentioned earlier, I keep my clock group information in a root variable called "clkgroups". clkgroups is indexed by clock group name, and each entry contains the clocks in that group:

```
--- ::clkgroups ---
clk0
  clk0
  clk3
  clk0_leaf
  clk3_leaf
clk1
  clk1
  clk1_leaf
iovirt
  iovirt
```

So, clkgroups(clk0) contains a list of "clk0 clk3 clk0_leaf clk3_leaf".

I have my own procs to use, display, and manipulate this array, so my implementation of "-group" involves a call to "&add_to_clkgroups":

```
  if {[info exists _group]} {
    &eval &add_to_clkgroups clkgroups $_group $_name
  }
```

To calculate uncertainties, I have procs called "&setup_uncertainty" and "&hold_uncertainty" that return the correct uncertainty based on the period (for &setup_uncertainty) and the stage of the process (larger uncertainty in synthesis, etc).

&create_clock has 3 uncertainty-related options:

- -setup_uncertainty <float> specifies the setup uncertainty
- -hold_uncertainty <float> specifies the hold uncertainty

- -default_uncertainty is a boolean that means to just use the standard procs to calculate the uncertainties

So, the setup uncertainty code looks like this:

```
  if {[info exists _setup_uncertainty] || $_default_uncertainty} {
    if {[info exists _setup_uncertainty]} {
      set setup_uncertainty $_setup_uncertainty
    } else {
      set setup_uncertainty [&setup_uncertainty $_period]
    }
    set setup_uncertaintyof($_name) [expr double($setup_uncertainty)]

    &eval set_clock_uncertainty -setup $setup_uncertainty [get_clocks $_name]
  }
```

Note that after the uncertainty is calculated, it is stored in the global setup_uncertaintyof array as well as being applied using the "set_clock_uncertainty" command.

Hold uncertainty is similar:

```
  if {[info exists _hold_uncertainty]} {
    &eval set_clock_uncertainty -hold $_hold_uncertainty [get_clocks $_name]
    set hold_uncertaintyof($_name) $_hold_uncertainty
  } elseif {$_default_uncertainty} {
    set hold_uncertaintyof($_name) [&hold_uncertainty]
    &eval set_clock_uncertainty -hold $hold_uncertaintyof($_name) [get_clocks
$_name]
  }
```

5.5 &create_generated_clocks

&create_generated_clocks is similar in structure to &create_clocks, except we're calculating and storing different things. In addition to storing period, source, setup uncertainty, and hold uncertainty, we're also going to store the master clock, the master source, the waveform, and the waveform modification (div2, for example).

Arguments are like create_generated_clock, with a few added in ***bold/italics***:

```
&define_proc_attributes &create_generated_clock \
  -info "Wrapper for create_generated_clock"  \
  -define_args {
  {-add "Same as create_generated_clock" "" boolean optional}
  {-combinational "Same as create_generated_clock" "" boolean optional}
  {-divide_by "Same as create_generated_clock" divide_by int optional}
  {-edge_shift "Same as create_generated_clock" edge_shift float optional}
  {-edges "Same as create_generated_clock" edges list optional}
  {-invert "Same as create_generated_clock" "" boolean optional}
  {-master_clock "Same as create_generated_clock" master_clock string
optional}
  {-multiply_by "Same as create_generated_clock" multiply_by int optional}
  {-name "Same as create_generated_clock" name string optional}
```

```
  {-source "Same as create_generated_clock" source string optional}
  {-setup_uncertainty "Setup uncertainty" setup_uncertainty float optional}
  {-hold_uncertainty "Hold uncertainty" hold_uncertainty float optional}
  {-default_uncertainty "Use &setup_uncertainty and &hold_uncertainty for
uncertainty" "" boolean optional 1}
  {-group "Group in clkgroups" group string optional}
  {-autogroup "Put in same group as master" "" boolean optional 1}
  {_src "Same as create_generated_clock" src string required ""}
}
```

Globalize things as usual:

```
proc &create_generated_clock { args } {
  global c
  global periodof
  global masterof
  global sourceof
  global msourceof
  global waveformof
  global wfmodof
  global setup_uncertaintyof
  global hold_uncertaintyof

  global clkgroups

  # Call the special parser.  Arg values will be set in simple variables
  # named with an _ prefix (replacing the - in the case of optional
variables).
  # Allows defaults to be given with the &define_proc_attributes call as well.
  #
  if {[&parse_proc_arguments -args $args results] != 1} { return }
```

source, master source, and master are arguments to the proc, so we can just save them away:

```
  set sourceof($_name) [get_object_name $_src]
  set msourceof($_name) [get_object_name $_source]
  set masterof($_name) $_master_clock
```

Note that the master source will likely be specified using a proc that looks up the sourceof entry for the master clock and returns it:

```
  set master clk0
  &eval create_generated_clock -divide_by 2 -name $_name -source [&source_of
$_master] -add -master $_master [get_pins ${_path}out0_reg/$c(qpin)
```

&create_generated_clock then calculates the period, waveform and wavefrom modification based on the master clock's period and the -divide_by, -multiply_by, etc options:

```
  # To avoid timing updates, pre-compute the period and waveform
  if {[info exists _divide_by]} {
    set periodof($_name) [expr [&period_of $_master_clock] * $_divide_by]
    set waveformof($_name) [list 0 [expr $periodof($_name) / 2]]
    set wfmodof($_name) "div($_divide_by)"
  } elseif {[info exists _multiply_by]} {
```

```
    set periodof($_name) [expr [&period_of $_master_clock] / $_multiply_by]
    set waveformof($_name) [list 0 [expr $periodof($_name) / 2]]
    set wfmodof($_name) "mult($_multiply_by)"
  } elseif {[info exists _combinational]} {
    set periodof($_name) [expr [&period_of $_master_clock] ]
    set waveformof($_name) [list 0 [expr $periodof($_name) / 2]]
    set wfmodof($_name) "div(1),combinational"
  } elseif {[info exists _edges]} {
    set periodof($_name) [expr [&period_of $_master_clock] * (([lindex $_edges
2] -1) / 2)]
    set hc [expr $periodof($_master_clock) / 2]
    set waveformof($_name) [list [expr ([lindex $_edges 0] - 1) * $hc] [expr
([lindex $_edges 1] - 1) * $hc] ]
    set wfmodof($_name) "edges( $_edges )"
  }
```

Then, the special options are removed and create_generated_clock is called:

```
  # Handle my args
  # Note: cannot use --, -_, or anything but -\w in pt for an option
  foreach {one_arg_opt_pattern} [list {-setup\S*} {-hold\S*}  {-group\S*}] {
    regsub -- "${one_arg_opt_pattern}\\s+\\S+" $args {} args
  }
  foreach {boolean_opt_pattern} [list {-default\S*} {-(no)?auto\S*}] {
    regsub -- "${boolean_opt_pattern}\\s*" $args {} args
  }

  # Do command with original args
  &eval uplevel 1 "create_generated_clock $args"
```

Again, group handling is flow-specific, but note that the default for boolean "-autogroup" is 1, so normally the master's group is used unless -group is specified explicitly.

```
  if {[info exists _group]} {
    &eval &add_to_clkgroups clkgroups $_group $_name
  } elseif {$_autogroup} {
    &eval &add_to_clkgroups clkgroups [&get_clkgroup $_master_clock] $_name
    &eval &duplicate_in_clkgroups -all -current $_master_clock -new $_name
  }
```

And uncertainty is done based on either the given value or the period of this new clock:

```
  if {[info exists _setup_uncertainty]} {
    &eval set_clock_uncertainty -setup $_setup_uncertainty [get_clocks $_name]
    set setup_uncertaintyof($_name) [expr double($_setup_uncertainty)]
  } elseif {$_default_uncertainty} {
    set setup_uncertaintyof($_name) [&setup_uncertainty [&get_clock_info -
period $_name]]
    &eval set_clock_uncertainty -setup $setup_uncertaintyof($_name)
[get_clocks $_name]
  }
  if {[info exists _hold_uncertainty]} {
    &eval set_clock_uncertainty -hold $_hold_uncertainty [get_clocks $_name]
    set hold_uncertaintyof($_name) $_hold_uncertainty
  } elseif {$_default_uncertainty} {
    set hold_uncertaintyof($_name) [&hold_uncertainty]
```

```
    &eval set_clock_uncertainty -hold $hold_uncertaintyof($_name) [get_clocks
$_name]
  }
```

6 Conclusion

7 Acknowledgements

The author would like to acknowledge the following individuals for their assistance:

8 References

(1) My Favorite DC/PT Tcl Tricks
Paul Zimmer
Synopsys Users Group 2003 San Jose
(available at www.zimmerdesignservices.com)

(2) There's a better way to do it! Simple DC/PT tricks that can change your life
Paul Zimmer
Synopsys Users Group 2010 San Jose
(available at www.zimmerdesignservices.com)

(3) Simplifying Constraints By Using More Generated Clocks
Stuart Hecht [SJH Clear Consulting LLC]
Synopsys Users Group 2009 San Jose
(available at SNUG website)

9 Appendix

9.1 &eval

```
# This proc can be used in front of any (non-"set") command to cause it to
echo
# the command and options before executing it.  It is similar to &cmd from
# Paul Zimmer's "My favorite dc/pt tcl tricks"
# paper (www.zimmerdesignservices.com)
proc &eval { args } {

  # extract my options, if any
  set _skiparg 0 ;# used for options with arguments to skip over the argument
  # default options
  set _interonly 0
  foreach _item $args {
    if {$_skiparg} {
      set _skiparg 0
    } else {
      if {[regexp {^--} $_item]} {
        # if matched, delete it
        set args [lreplace $args 0 0]
        # now parse it
        switch -glob -- $_item {
          --interonly {
            set _interonly 1
          }
          --* {
            &err "[&myname] got unrecognized argument $_item"
            return
          }
        }
      } else {
        break
      }
    }
  }

  if {$_interonly && (![info exists ::env(DC_INTER)] || [is_false
$::env(DC_INTER)])} {
    echo "Skipping command \"$args\" because interactive mode is not set."
  } else {
    # get rid of any embedded cr's in args
    regsub -all {\n} $args {} args
    # echo result
    if {$::tool eq "rtl_compiler"} {
      set _newargs ""
      foreach _arg $args {
        #puts "got here with arg \"$_arg\" and llength [llength $_arg]"
        if {[info exists ::pz_command_listmax] && [llength $_arg] >
$::pz_command_listmax} {
          #puts "got there"
          #regsub $ arg $ newargs "\{ [&list coll -verbose -limit
$::pz_command_listmax $_arg] \}" _newargs
          #append _newargs [string range $args 0 [expr [string first $_arg
$args] - 1]]
```

```
          #append _newargs [&list_coll -verbose -limit $::pz_command_listmax
$_arg]
          #append _newargs [string range $args  [expr [string first $_arg
$args] + [string length $_arg]] [string length $args]]
          append _newargs "{[&list_coll -verbose -limit $::pz_command_listmax
$_arg]} "
          #puts "_newargs is now $_newargs"
        } else {
          append _newargs "$_arg "
          #puts "_newargs is now $_newargs"
        }
      }
      set _absdate [clock seconds] ;
      if {![info exists ::_old_cmd_absdate]} {set ::_old_cmd_absdate
$_absdate}
      if {[info exists ::_cmd_timestamp_on] && $::_cmd_timestamp_on} {
        redirect -var _date date
        regsub {\n} $_date {} _date
        set _ts " ; $_date (abs: ${_absdate} ) (time_since_last_cmd: [expr
$_absdate - $::_old_cmd_absdate] )"
      } else {
        set _ts ""
      }
    } else {
      set _newargs $args
      foreach _arg $args {
        if {[regexp {^_sel[0-9]+} $_arg _junk]} {
          #echo "found collection $_arg"
          if {[info exists ::_command_listmax]} {
            regsub $_arg $_newargs "\{ [&list_coll -verbose -limit
$::_command_listmax $_arg] \}" _newargs
          } else {
            regsub $_arg $_newargs "\{ [&list_coll $_arg] \}" _newargs
          }
        }
      }
      set _absdate [clock seconds] ;
      if {![info exists ::_old_cmd_absdate]} {set ::_old_cmd_absdate
$_absdate}
      if {[info exists ::_cmd_timestamp_on] && $::_cmd_timestamp_on} {
        set _ts " ; [date] (abs: ${_absdate} ) (time_since_last_cmd: [expr
$_absdate - $::_old_cmd_absdate] )"
      } else {
        set _ts ""
      }
    }



    echo "Doing command: $_newargs $_ts"
    set ::_old_cmd_absdate $_absdate
    # Do command
    uplevel 1 $args
  }
}

# Note that this is for help and syntax highlight parsing only -
parse_proc_argument is never called...
#define_proc_attributes &eval -default_options_to_null -info "Echo a command
and then execute it." \
```

```
#  -define_args \
#  {
#      {--interonly "Only execute if in interactive mode" "" boolean optional}
#  }
```

9.2 &define_proc_attributes / &parse_proc_arguments

The code for &define_proc_attributes and &parse_proc_arguments is different in DC/PT versus RC. So, it checks a global variable "tool" first.

```
if {$::tool eq "rtl_compiler"} {
  # &define_proc_attributes
  # This proc mimics the behavior of my &define_proc_attributes proc for
dc/pt, which is
  # a wrapper around dc/pt's define_proc_attributes command.  In rc, it works
a little differently.
  # The proc executes when the procs are being read in, and creates entries in
2 global
  # arrays:
  #  _parse_options
  #  _parse_defaults
  #
  # The first contains the full "parse_options" command that rc will execute
for that proc.
  # The second contains code that sets the defaults in the same way that my
dc/pt flow does
  # it (-var set to _var, booleans default to 0, others to "").
  #
  # The proc &parse_proc_arguments then calls these array entries.
  #
  # This allows me to port over procs from my dc/pt collection more easily.
  proc &define_proc_attributes { args } {
    global _parse_options
    global _parse_defaults
    #puts $args
    # pop proc name off the front
    set procname [&pop args]
    #puts "procname $procname"
    set _parse_options($procname) "\
    switch -- \[ \\
    parse_options $procname file_var \$args \\"
    set _parse_defaults($procname) ""

    set _default_options_to_null 0
    while {[llength $args] > 0} {
      set arg [&pop args]
      if {[regexp {^-} $arg]} {
        switch [lindex $arg 0] {
          -default_options_to_null {
            set _default_options_to_null 1
          }
          -info {
            append arg " " [&pop args]
            #puts "info: $arg ..."
          }
          -define_args {
            set definitions [&pop args]
```

```
            #puts "definitions: $definitions ..."
            # split this by line
            set raw_options [split $definitions "\n"]
            # rc insists that - options precede other options.  So, reorder
the options accordingly.
            set options [list]
            set endoptions [list]
            foreach option $raw_options {
              if {[regexp {\{-} $option]} {
                lappend options $option
                #echo "option is $option"
              } else {
                lappend endoptions $option
              }
            }
            set options [concat $options $endoptions]

            # Now loop through the options and create an entry for each in the
parse_options cmd
            # Entries look like this:
            #     {-unless sos Only output if arg is false} _unless \
            #     {-timestamp bos Use absolute timestamp} _timestamp \
            #     {sos Message to print} _message \
            # Note that non-dash options drop the first field

            foreach option $options {
              if {[llength $option] > 0} {
                # turn it into a list
                eval "set descriptor $option"
                # parse the list by trickery
                foreach {optname help junk type reqopt default} $descriptor {}
                # Can't tell the difference btw no descriptor and a descriptor
of "".  Unset if no descriptor
                if {[llength $descriptor] == 5} {
                  unset default
                }

                # If this is not a dash option, leave out the first entry in
the line.
                # (see comment above)
                # To make sure no blanks after the opening curly brace, we'll
resort to
                # creating a variable with or without the first word and the
blank.
                if {![regexp {^-} $optname]} {
                  set firstword ""
                } else {
                  regsub -- {-} $optname {} ;# strip extra - if present
                  set firstword "$optname "
                }

                # build the desc string
                # first char - type
                switch -- $type {
                  "boolean" {set desc "b"}
                  "string" {set desc "s"}
                  "float" {set desc "f"}
                  "int" {set desc "n"}
                  "integer" {set desc "n"}
                  "int" {set desc "n"}
```

```
                  "list" {set desc "s"}
                  default {&err "Unrecognized type $type in [&myname]"}
                }
                # second char - required/optional
                switch -- $reqopt {
                  "required" {set desc "${desc}r"}
                  "optional" {set desc "${desc}o"}
                  default {&err "Unrecognized reqopt $reqopt in [&myname]"}
                }
                # third character - single/multiple
                # don't remember how multi stuff is done in synopsys, just
assume single
                set desc "${desc}s"
                # this gives " The 'som' and 'srm' do not support flagged
options."
                #if {$type eq "list"} {
                #  set desc "${desc}m"
                #} else {
                #  set desc "${desc}s"
                #}

                # The last field is the name of the variable.  By convention
(mine), for
                # dash options, this is the option name with - replaced by _
                regsub -- {--?} $optname {_} varname

                # Now create the line in _parse_options($procname)
                set _parse_options($procname) "$_parse_options($procname)
     \{${firstword}$desc $help\} $varname \\"

                # Create the default set command in _parse_defaults($procname)
                # NOTE: ![string compare] means it compares true!
                #
                # If the default value is set, use that
                if {[info exists default]} {
                  set _parse_defaults($procname) "$_parse_defaults($procname)
if {!\[string compare \$$varname \"\"\]} {set $varname \"$default\"}"
                # otherwise, if it's boolean, set it to 0
                } elseif {$type eq "boolean"} {
                  set _parse_defaults($procname) "$_parse_defaults($procname)
if {!\[string compare \$$varname \"\"\]} {set $varname 0}"

                # otherwise, if its a float, force it to be a float unless it
is a null string
                } elseif {$type eq "float"} {
                  set _parse_defaults($procname) "$_parse_defaults($procname)
if {\[string compare \$$varname \"\"\]} {set $varname \[expr
double(\$$varname)\]}"
                  # If it's optional, add the "unset" code like below
                  if {$reqopt ne "required"} {
                    if {$_default_options_to_null} {
                      set _parse_defaults($procname)
"$_parse_defaults($procname)
if {!\[string compare \$$varname \"\"\]} {set $varname \"\"}"
                    } else {
                      set _parse_defaults($procname)
"$_parse_defaults($procname)
if {!\[string compare \$$varname \"\"\]} {unset $varname}"
                    }
                  }
```

```
                # If it's a string, set it to null string if -
default_options_to_null is set
                # RC's own parser does this, so undo it
                } elseif {($reqopt ne "required") && ($type eq "string" ||
$type eq "list" || $type eq "int")} {
                  if {$_default_options_to_null} {
                    set _parse_defaults($procname)
"$_parse_defaults($procname)
if {![string compare \$$varname \"\"\]} {set $varname \"\"}"
                  } else {
                    set _parse_defaults($procname)
"$_parse_defaults($procname)
if {![string compare \$$varname \"\"\]} {unset $varname}"
                  }
                } else {
                }

                # Handle booleans with default of 1
                # - accept -no option
                if {$type eq "boolean" && [info exists default] && $default ==
1} {
                  # Create the line in _parse_options($procname)
                  #puts "got here for $procname"
                  regsub -- {-} $firstword {-no} firstword
                  regsub -- {_} $varname {_no} varname_no
                  set _parse_options($procname) "$_parse_options($procname)
      \{${firstword}$desc Turn off $help\} ${varname_no} \\"

                  set _parse_defaults($procname) "$_parse_defaults($procname)
set $varname $default"
                  set _parse_defaults($procname) "$_parse_defaults($procname)
if {\$$varname_no == 1} {set $varname 0}"
                }

              }
            }
          }
        }
      }
    }
    set _parse_options($procname) "$_parse_options($procname)
    ] {
      -2 {return}
      0 {return -code error}
    }
  "
  }

  # &parse_proc_arguments
  # This proc mimics the behavior of my &parse_proc_arguments proc for dc/pt,
which is
  # a wrapper around dc/pt's parse_proc_arguments that handles defaults
defined in
  # the corresponding &default_proc_attributes.
  # Most of the heavy lifting is done by &default proc attributes, and its
results are
  # executed here.
  proc &parse_proc_arguments { args } {
    set calling_proc [uplevel 1 &myname]
```

```
    uplevel 1 $::_parse_options($calling_proc)
    uplevel 1 $::_parse_defaults($calling_proc)

    # This allows the calling proc to determine whether this proc completed.
    return 1
  }

} else {
  # &define_proc_attributes
  #   wrapper for define_proc_attributes that handles defaults
  #   defaults are stored as "set" commands in global arrays _proc_defaults
  #   and _proc_default_booleans.  These entries are then executed by
  #   &parse_proc_arguments when it executes.
  proc &define_proc_attributes { args } {

    global _proc_defaults
    global _proc_default_booleans

    # Body of process

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

    #echo "Defining proc args for $_proc_name"
    # Check for -default_options_to_null and remove it if found
    set _default_opts_arg_index [lsearch $args {-defa*}]
    if {$_default_opts_arg_index != -1} {
      set _default_options_to_null 1
      set args [lreplace $args $_default_opts_arg_index
$_default_opts_arg_index] ; # crush it out
    } else {
      set _default_options_to_null 0
    }

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

    set _cmd "" ;# init command to null
    array unset _proc_default_booleans $_proc_name ;# rm any previous results
    # March through arg defs and find any optional ones with defaults
    set _new_arg_defs {}
    foreach _def $_arg_defs {
      if {[lindex $_def 3] == "boolean"} {
        # Booleans always default to 0 unless otherwise specified
        if {[lindex $_def 4] == "optional" && [llength $_def] > 5} {
          # Found one.  Grab default, then crush it out
          set _default [lindex $_def 5]
          set _def [lreplace $_def 5 5]
          if {$_default == 1} {
            set _nodef $_def  ;# Flag it to get a "-no" version
          }
          set _def [lreplace $_def 1 1 "[lindex $_def 1] - defaults to
$_default"]
        } else {
          set _default 0
        }
        # Now append the set to $_cmd
```

```
        set _name [lindex $_def 0]
        set _cmd "set results($_name) { $_default } ; $_cmd"

        # Figure out the "-no" version.
        # Only do this if the default was 1 (indicated by _nodef having been
        # created above).
        if {[info exists _nodef]} {
          set _noname [regsub {^-} $_name {no}]
          # If there isn't such an argument
          # specified, create it and keep a list of these for
          # &proc_parse_arguments
          if {[string first "-${_noname}" $_arg_defs_string] == -1} {
           set _cmd "set results(-$_noname) { 0 } ; $_cmd"
           set _nodef [lreplace $_nodef 0 0 "-${_noname}"]
           set _nodef [lreplace $_nodef 1 1 "*DON'T* [lindex $_nodef 1]"]
            append _proc_default_booleans($_proc_name) " ${_noname}"
            regsub {^ } $_proc_default_booleans($_proc_name) {}
_proc_default_booleans($_proc_name)
          }
        }

      # Handle defaults
      } 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"

      # Handle -default_options_to_null
      } elseif {$_default_options_to_null && [lindex $_def 4] == "optional" &&
[llength $_def] <= 5} {
        set _name [lindex $_def 0]
        set _cmd "set results($_name) \"\" ; $_cmd"
      }

      # Rebuild the arg defs as we go
      lappend _new_arg_defs $_def
      if {[info exists _nodef]} {
        lappend _new_arg_defs $_nodef
      }
      #echo "new defs: $_new_arg_defs"
    }

    # 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
  #  echo "define_proc_attributes $new_args\n";
    eval define_proc_attributes $new_args
  }
```

```
  define_proc_attributes &define_proc_attributes -info "Wrapper for
define_proc_attributes and allows optional arguments to be defaulted." \
    -define_args \
    {
    }

  proc &parse_proc_arguments { args } {

    global _proc_defaults
    global _proc_default_booleans
    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
    #echo "parse_proc_arguments $args"
    uplevel 1 parse_proc_arguments $args
    #parray results
    #echo "res is $res"

    # 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
    }

    # Handle "-no{flag}" version of booleans
    if {[info exists _proc_default_booleans($_proc_name)]} {
      foreach _noname [split $_proc_default_booleans($_proc_name) { }] {
        set _name [regsub {^no} $_noname {}]
        set _cmd "if \{\$_$_noname\} \{set _${_name} 0\}"
        #echo "Setting $_noname \"$_cmd\""
        uplevel 1 $_cmd
      }
    }
```

```
    # This allows the calling proc to determine whether this proc completed.
    # If -help was specified, dc/pt tcl will bail out on the call to
    # parse_proc_arguments above, and this proc will never get here.
    # The calling proc can detect this by checking for a non-"1" return value.
    return 1
  }



}
```

9.3 &create_clock

This is just to give a flavor of what you might do. It uses a bunch of stuff from my flow that isn't included here.

```
# Wrapper for create_clock to capture data into arrays
proc &create_clock { args } {
  global c
  global periodof
  global waveformof
  global sourceof
  global setup_uncertaintyof
  global hold_uncertaintyof

  global clkgroups

  # Call the special parser.  Arg values will be set in simple variables
  # named with an _ prefix (replacing the - in the case of optional
variables).
  # Allows defaults to be given with the &define proc attributes call as well.
  #
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  # Make sure period is float
  set _period [expr double($_period)]

  # Save data in global arrays
  set periodof($_name) $_period
  if {[info exists _waveform]} {
    set waveformof($_name) $_waveform
  } else {
    set waveformof($_name) [list 0 [expr $_period/2]]
  }

  set sourceof($_name) [get_object_name $_src]

  # Handle my args
  foreach {one_arg_opt_pattern} [list {-setup\S*} {-hold\S*} {-group\S*}] {
    regsub -- "${one_arg_opt_pattern}\\s+\\S+" $args {} args
  }
  foreach {boolean_opt_pattern} [list {-default\S*}] {
    regsub -- "${boolean_opt_pattern}\\s*" $args {} args
  }

  # Do command with original args, minus mine
  &eval uplevel 1 "create_clock $args"
```

```
  # Apply my args
  if {[info exists _group]} {
    &eval &add_to_clkgroups clkgroups $_group $_name
  }

  if {[info exists _setup_uncertainty] || $_default_uncertainty} {
    if {[info exists _setup_uncertainty]} {
      set setup_uncertainty $_setup_uncertainty
    } else {
      set setup_uncertainty [&setup_uncertainty $_period]
    }
    set setup_uncertaintyof($_name) [expr double($setup_uncertainty)]

    &eval set_clock_uncertainty -setup $setup_uncertainty [get_clocks $_name]
  }

  if {[info exists _hold_uncertainty]} {
    &eval set_clock_uncertainty -hold $_hold_uncertainty [get_clocks $_name]
    set hold_uncertaintyof($_name) $_hold_uncertainty
  } elseif {$_default_uncertainty} {
    set hold_uncertaintyof($_name) [&hold_uncertainty]
    &eval set_clock_uncertainty -hold $hold_uncertaintyof($_name) [get_clocks
$_name]
  }

}
&define_proc_attributes &create_clock \
  -info "Wrapper for create_clock"  \
  -define_args {
  {-add "Same as create_clock" "" boolean optional}
  {-name "Same as create_clock" name string optional}
  {-period "Same as create_clock" period float optional}
  {-waveform "Same as create_clock" waveform list optional}
  {-setup_uncertainty "Setup uncertainty" setup_uncertainty float optional}
  {-hold_uncertainty "Hold uncertainty" hold_uncertainty float optional}
  {-default uncertainty "Use &setup uncertainty and &hold uncertainty for
uncertainty" "" boolean optional}
  {-group "Group in clkgroups" group string optional}
  {_src "Same as create_clock" src string optional ""}
}
```

9.4 &create_generated_clock

As above, just for flavor.

```
# Wrapper for create_generated_clock to capture data into arrays.  This helps
to avoid
# timing updates.
proc &create_generated_clock { args } {
  global c
  global periodof
  global masterof
  global sourceof
  global msourceof
  global waveformof
  global wfmodof
```

```
  global setup_uncertaintyof
  global hold_uncertaintyof

  global clkgroups

  # Call the special parser.  Arg values will be set in simple variables
  # named with an _ prefix (replacing the - in the case of optional
variables).
  # Allows defaults to be given with the &define_proc_attributes call as well.
  #
  if {[&parse_proc_arguments -args $args results] != 1} { return }

  # Since reports will use full path, do this here as well
  set sourceof($_name) [get_object_name $_src]
  set msourceof($_name) [get_object_name $_source]
  set masterof($_name) $_master_clock

  # To avoid timing updates, pre-compute the period and waveform
  if {[info exists _divide_by]} {
    set periodof($_name) [expr [&period_of $_master_clock] * $_divide_by]
    set waveformof($_name) [list 0 [expr $periodof($_name) / 2]]
    set wfmodof($_name) "div($_divide_by)"
  } elseif {[info exists _multiply_by]} {
    set periodof($_name) [expr [&period_of $_master_clock] / $_multiply_by]
    set waveformof($_name) [list 0 [expr $periodof($_name) / 2]]
    set wfmodof($_name) "mult($_multiply_by)"
  } elseif {[info exists _combinational]} {
    set periodof($_name) [expr [&period_of $_master_clock] ]
    set waveformof($_name) [list 0 [expr $periodof($_name) / 2]]
    set wfmodof($_name) "div(1),combinational"
  } elseif {[info exists _edges]} {
    set periodof($_name) [expr [&period_of $_master_clock] * (([lindex $_edges
2] -1) / 2)]
    set hc [expr $periodof($_master_clock) / 2]
    set waveformof($_name) [list [expr ([lindex $_edges 0] - 1) * $hc] [expr
([lindex $_edges 1] - 1) * $hc] ]
    set wfmodof($_name) "edges( $_edges )"
  }


  # Handle my args
  # Note: cannot use --, -_, or anything but -\w in pt for an option
  foreach {one_arg_opt_pattern} [list {-setup\S*} {-hold\S*}  {-group\S*}] {
    regsub -- "${one_arg_opt_pattern}\\s+\\S+" $args {} args
  }
  foreach {boolean_opt_pattern} [list {-default\S*} {-(no)?auto\S*}] {
    regsub -- "${boolean_opt_pattern}\\s*" $args {} args
  }

  # Do command with original args
  &eval uplevel 1 "create_generated_clock $args"

  # Apply my args
  if {[info exists _group]} {
    &eval &add_to_clkgroups clkgroups $_group $_name
  } elseif {$_autogroup} {
    &eval &add_to_clkgroups clkgroups [&get_clkgroup $_master_clock] $_name
    &eval &duplicate_in_clkgroups -all -current $_master_clock -new $_name
  }
```

```
  if {[info exists _setup_uncertainty]} {
    &eval set_clock_uncertainty -setup $_setup_uncertainty [get_clocks $_name]
    set setup_uncertaintyof($_name) [expr double($_setup_uncertainty)]
  } elseif {$_default_uncertainty} {
    set setup_uncertaintyof($_name) [&setup_uncertainty [&get_clock_info -
period $_name]]
    &eval set_clock_uncertainty -setup $setup_uncertaintyof($_name)
[get_clocks $_name]
  }
  if {[info exists _hold_uncertainty]} {
    &eval set_clock_uncertainty -hold $_hold_uncertainty [get_clocks $_name]
    set hold_uncertaintyof($_name) $_hold_uncertainty
  } elseif {$_default_uncertainty} {
    set hold_uncertaintyof($_name) [&hold_uncertainty]
    &eval set_clock_uncertainty -hold $hold_uncertaintyof($_name) [get_clocks
$_name]
  }

}

&define_proc_attributes &create_generated_clock \
  -info "Wrapper for create_generated_clock"  \
  -define_args {
  {-add "Same as create_generated_clock" "" boolean optional}
  {-combinational "Same as create_generated_clock" "" boolean optional}
  {-divide_by "Same as create_generated_clock" divide_by int optional}
  {-edge_shift "Same as create_generated_clock" edge_shift float optional}
  {-edges "Same as create_generated_clock" edges list optional}
  {-invert "Same as create_generated_clock" "" boolean optional}
  {-master_clock "Same as create_generated_clock" master_clock string
optional}
  {-multiply_by "Same as create_generated_clock" multiply_by int optional}
  {-name "Same as create_generated_clock" name string optional}
  {-source "Same as create_generated_clock" source string optional}
  {-setup_uncertainty "Setup uncertainty" setup_uncertainty float optional}
  {-hold_uncertainty "Hold uncertainty" hold_uncertainty float optional}
  {-default_uncertainty "Use &setup_uncertainty and &hold_uncertainty for
uncertainty" "" boolean optional 1}
  {-group "Group in clkgroups" group string optional}
  {-autogroup "Put in same group as master" "" boolean optional 1}
  {_src "Same as create_generated_clock" src string required ""}
}
```

9.5 vname/dc:: namespace, etc

```
# Set up dummy dc:: version of commands for compatibility with rc code
namespace eval dc {
  foreach cmd [list \
    add_to_collection \
    all_clocks \
    all_fanin \
    all_fanout \
    all_inputs \
    all_outputs \
    all_registers \
    append_to_collection \
    check_override \
```

```
compare_collection \
copy_collection \
create_clock \
create_generated_clock \
current_design \
current_instance \
filter_collection \
foreach_in_collection \
get_cell \
get_cells \
get_clock \
get_clocks \
get_design \
get_designs \
getenv \
get_generated_clocks \
get_lib \
get_lib_cell \
get_lib_cells \
get_lib_pin \
get_lib_pins \
get_libs \
get_lib_timing_arcs \
get_net \
get_nets \
get_object_name \
get_path_groups \
get_pin \
get_pins \
get_port \
get_ports \
group_path \
index_collection \
remove_clock_gating_check \
remove_clock_latency \
remove_disable_clock_gating_check \
remove_from_collection \
remove_generated_clock \
remove_ideal_net \
remove_ideal_network \
remove_input_delay \
remove_output_delay \
set_case_analysis \
set_clock_gating_check \
set_clock_groups \
set_clock_latency \
set_clock_sense \
set_clock_skew \
set_clock_transition \
set_clock_uncertainty \
set_data_check \
set_disable_clock_gating_check \
set_disable_timing \
set_dont_touch \
set_dont_touch_network \
set_dont_use \
set_drive \
set_driving_cell \
```

```
    set_equal \
    set_false_path \
    set_fanout_load \
    set_hierarchy_separator \
    set_ideal_net \
    set_ideal_network \
    set_input_delay \
    set_input_transition \
    set_lib_pin \
    set_lib_pins \
    set_load \
    set_load_unit \
    set_logic_dc \
    set_logic_one \
    set_logic_zero \
    set_max_capacitance \
    set_max_delay \
    set_max_dynamic_power \
    set_max_fanout \
    set_max_leakage_power \
    set_max_time_borrow \
    set_max_transition \
    set_min_delay \
    set_multicycle_path \
    set_operating_conditions \
    set_opposite \
    set_output_delay \
    set_path_adjust \
    set_port_fanout_number \
    set_time_unit \
    set_timing_derate \
    set_unconnected \
    set_units \
    set_wire_load_mode \
    set_wire_load_model \
    set_wire_load_selection_group \
    sizeof_collection \
    sort_collection \
  ] {
    eval "proc $cmd args \{ uplevel 1 $cmd \$args \}"
  }
}

proc vname args {
  # vname in RC returns the verilog name.  The same thing here, except that we
might
  # have to convert from a collection to a list
  if {[regexp {^_sel\d+} $args]} {
    return [&list_coll $args]
  } else {
    return $args
  }
}

proc set_time_unit args {
}

proc dirname args {
  return [regsub {/[^\/]+$} $args {}]
```

```
}

proc basename args {
  return [regsub {^.*/} $args {}]
}
```