dc_perl: Enhancing dc_shell using a Perl wrapper

Steve Golson

Trilobyte Systems, 33 Sunset Road, Carlisle MA 01741
Phone: +1.508.369.9669
Fax: +1.508.371.9964
Email: sgolson@trilobyte.com

***Abstract*: Is there a command that you wish dc_shell had?**

By using the Perl interpreter as a "wrapper" around dc_shell, powerful extensions to dc_shell can be created. dc_shell commands can be generated by Perl, and the results analyzed by Perl in real time (not post-processed). Further dc_shell commands can be algorithmically generated by Perl based on the given results.

The user interface is just like dc_shell, but with user-defined extensions. This approach is particularly suited for complex synthesis problems that currently require lots of post-processing or tedious human analysis.

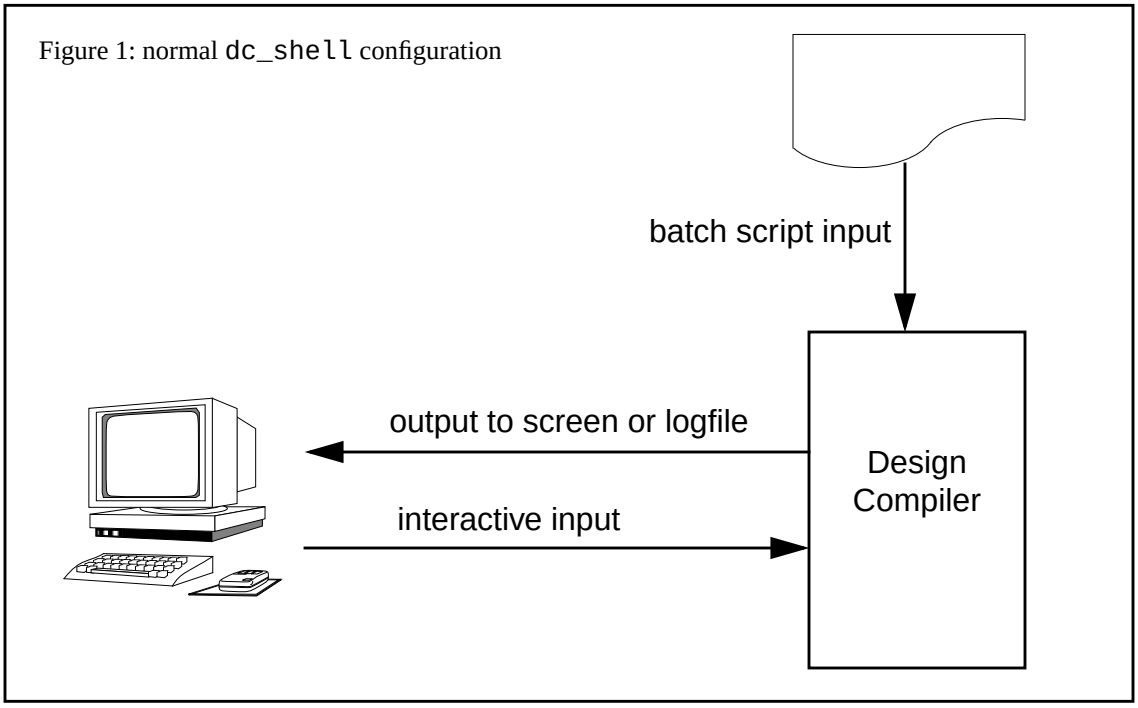
The problem

dc_shell provides a simple programming interface to Design Compiler (see Figure 1). However for many complex synthesis tasks it has significant limitations, including:

- no subroutines
- no variable scoping
- limited arithmetic and logical operations
- primitive list processing
- inflexible extensibility (sh, execute)
- primitive pattern matching (regular expressions)

What we need is a simple, flexible, and above all *powerful* way to extend dc_shell. These extensions should work both within scripts and interactively. All existing dc_shell scripts should work without modification.

Figure 1: normal dc_shell configuration

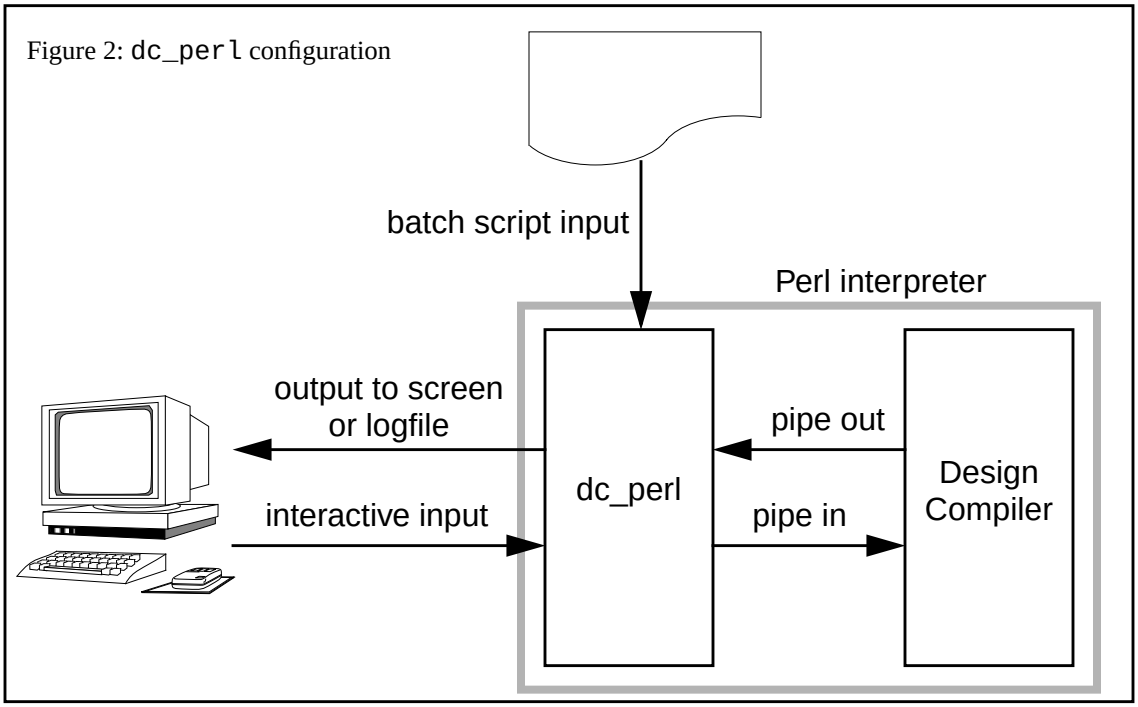


A solution

What we need is for dc_shell to be more like Perl! Perl, the Practical Extraction and Report Language[1], is (to quote the manpage) an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. Best of all, it's freely available and freely redistributable.

A Perl program called dc_perl has been developed. It parses a stream of commands entered interactively or via batch files, and determines which commands are Perl and which are dc_shell. The Perl commands are evaluated directly, and the dc_shell commands are passed via pipes to an actual dc_shell process running as a child under Perl. Output is controlled by the dc_perl wrapper, so dc_shell command output can be filtered and processed before being printed to the main dc_perl logfile (see Figure 2).

Figure 2: dc_perl configuration



dc_perl scripts can be thought of in several ways: as dc_shell scripts with a few "extensions", or as Perl scripts that occasionally call dc_shell, or anything in between.

1. Or depending on your level of experience, the Pathologically Eclectic Rubbish Lister.

Invoking dc_perl

dc_perl is invoked from your usual UNIX command prompt:

```
unix% dc_perl
dc_perl version 0.0
                        DC Professional (TM)
                           DC Expert (TM)
                         HDL Compiler (TM)

                    Version v3.4b -- Apr 01, 1996
               Copyright (c) 1988-1995 by Synopsys, Inc.
                         ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys, I
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

Initializing...
dc_perl>
```

This looks like a normal dc_shell session, except a dc_perl version number is displayed and the user prompt is changed to dc_perl>.

The -f command-line switch can be used to specify a dc_perl batch script. All other dc_shell command-line switches are passed through to the dc_shell process.

dc_perl commands

All dc_shell commands work normally. These keywords are used to switch into the Perl interpreter:

```
&begin_perl;
# all lines between these two are evaluated by the perl interpreter
&end_perl;
```

The following Perl commands are predefined for interfacing with dc_shell:

`&get_dc_shell_variable("variable");`

Gets the value of a dc_shell variable. Returns a list or scalar depending on the type of dc_shell variable.

`&set_dc_shell_variable("variable", value);`

Sets the value of the specified dc_shell variable. If value is a Perl list, then variable is assigned as a dc_shell list.

`&dc_shell_cmd("command");`

Executes the given dc_shell command string. The output is printed to standard output.

`&get_dc_shell_cmd("command");`

Executes the given dc_shell command string, and returns the output instead of printing it. This is used when you wish to filter or parse the output of the command.

Any non-comment line that has "&" as the first non-whitespace character, and ends with a ";" is assumed to be a Perl function and is evaluated by the Perl interpreter (i.e. if it looks like Perl it is assumed to be Perl). This allows user-defined Perl subroutines to be invoked directly without using the &begin_perl and &end_perl constructs.

Example: How to get the cell name when you have the pin name

In dc_shell if you have a pin name (from the all_connected() command, for example) you might want to get the corresponding cell name. This requires a simple regular expression substitution to strip off the last hierarchical element of the pin name. However it is awkward and difficult to do this in dc_shell.[2]

Here is a dc_perl script that generates a list of cell names from a list of pin names.

```
/* ...dc_shell commands... */
/* the dc_shell variable is called mypins */

&begin_perl;
# these are perl commands
@list = &get_dc_shell_variable("mypins");
# strip off the trailing /... from each element in the list
grep { s?/[^/]*$?? } @list;
# create the dc_shell variable
&set_dc_shell_variable("mycells", @list);
&end_perl;

/* the dc_shell variable mycells has the list in it */
/* ...more dc_shell commands... */
```

Alternatively this can all be defined as a Perl subroutine:

```
/* ...dc_shell commands... */
&begin_perl;
sub getcells {
  my ($cellvar, $pinvar) = @_;
  my @list = &get_dc_shell_variable($pinvar);
  grep { s?/[^/]*$?? } @list;
  &set_dc_shell_variable($cellvar, @list);
  }
&end_perl;
/* ...more dc_shell commands... */
```

Now this subroutine can be invoked directly from dc_shell. Here is how you might use this interactively:

```
dc_perl> list mylist
mylist = {"a/b/cde", "f/g/h/ijk", "l", "m/n/op"}
1

dc_perl> &getcells("newlist", "mylist");

dc_perl> list newlist
{"a/b", "f/g/h", "l", "m/n"}
1
```

---

2. All right, if you really must know, here's an alias that will do it. The variable names are hard-coded. Yuck.

```
alias get_thecells " \
  sh \" \
    (echo -n \\\"thecells = \\\" \\\; \
     echo \"thepins\") | \
    sed -e 's?/[^,}/][^,}/]*[,}]?,?g' -e 's/,$/}/' > tmp \" \; \
  include tmp \; \
  sh /bin/rm tmp "
```


Example: Extracting the slack from a timing report

Analyzing timing reports is a common and sometimes tedious task. With dc_perl we can automatically parse the timing report and extract values from it. For example, if the slack is below a certain amount then we might wish to do a further compile step on this module.

Here is a dc_perl subroutine which extracts the slack value from a timing report:

```
sub get_slack {
  $_ = &get_dc_shell_cmd("report_timing");
  m/ slack\s+  # keyword slack
     \S+\s+    # followed by one more word
     (\S+)     # then the value we want
     /x || warn "got no match";
  &set_dc_shell_variable("dc_shell_status",$1);
  }
```

Running report_timing directly we get

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : gf_mult
Version: v3.4b
Date   : Fri Jan 17 09:22:11 1997
****************************************

Operating Conditions: typical   Library: access05_5v
Wire Loading Model Mode: top

  Startpoint: in_b[0] (input port)
  Endpoint: out[0] (output port)
  Path Group: default
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  input external delay                     0.00       0.00 r
  in_b[0] (in)                             0.00       0.00 r
  U156/Y (NAND2X2)                         0.11       0.11 f
  …
  U162/Y (XOR2X1)                          0.39       3.07 f
  out[0] (out)                             0.00       3.07 f
  data arrival time                                   3.07

  max_delay                                8.00       8.00
  output external delay                    0.00       8.00
  data required time                                  8.00
  -----------------------------------------------------------
  data required time                                  8.00
  data arrival time                                  -3.07
  -----------------------------------------------------------
  slack (MET)                                         4.93
```


Instead we can invoke &get_slack which will run report_timing for us and parse the output, leaving the slack value in dc_shell_status:

```
dc_perl> &get_slack();
4.930000

dc_perl> list dc_shell_status
dc_shell_status = 4.930000
1
```

A simple modification to get_slack would allow report_timing arguments to be passed through.

Future work

A sophisticated analysis of timing reports would allow dc_perl to generate true timing budgets for a hierarchical design, without the limitations of characterize.

Complex automated synthesis techniques are made feasible with the powerful combination of Perl and dc_shell.

Availability

All dc_perl scripts can be retrieved via anonymous ftp from

```
ftp://ftp.ultranet.com/pub/sgolson/dc_perl
```


Please contact the author if you have any comments or suggestions regarding dc_perl.

