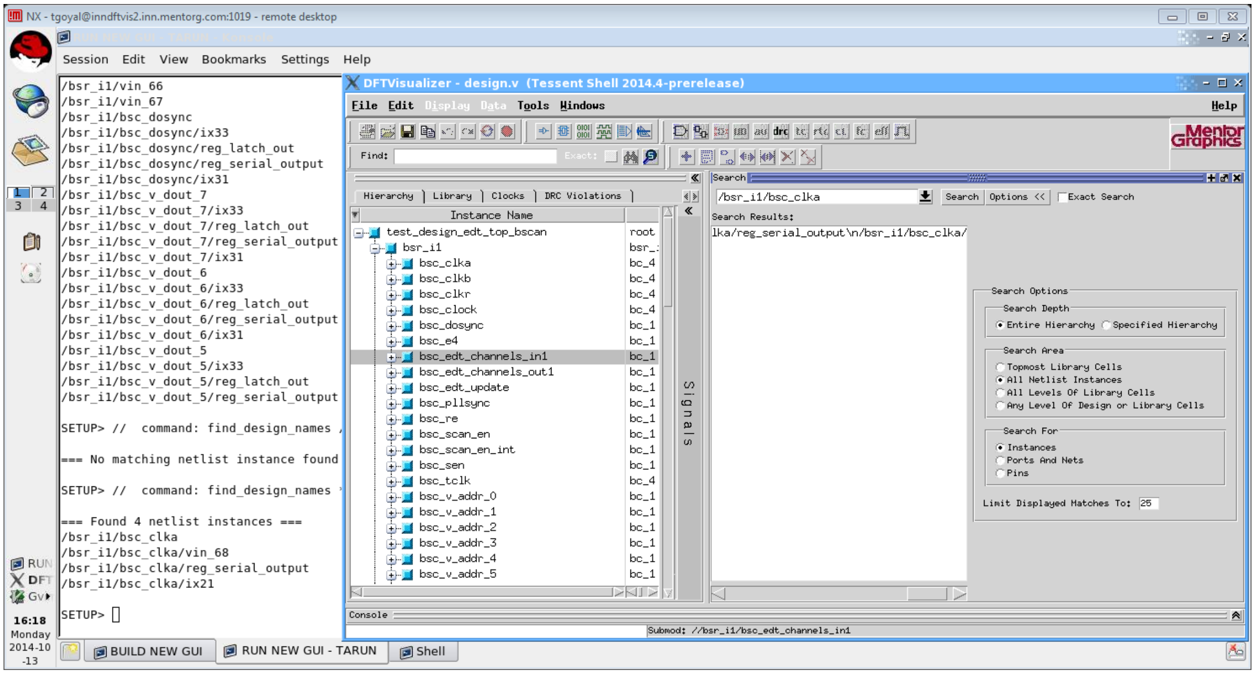
Authors: Tarun Goyal, Roshni Lalwani
tarun_goyal@mentor.com
roshni_lalwani@mentor.com

Title: Co-existence of a GUI and the main terminal: How was it achieved with the DFTVisualizer GUI with TCL/TK as the implementation language and the related changes?

Abstract: Most of the tools can either be invoked in GUI or non-GUI mode but not both at the same time. However, with DFTVisualizer we have been able to make the non-GUI shell and DFTVisualizer co-exit bringing about significant advantages to a customer, especially when the GUI is used for debugging purposes. However, such a change in user interaction methodology brings with it a set of challenges of interaction with the console and managing data through TCL interpreters. This paper will analyze such issues and present an approach to accomplish the co-existence well.


NX - tgoyal@inndftvis2.inn.mentorg.com:1019 - remote desktop
Session Edit View Bookmarks Settings Help
DFTVisualizer - design.v (Tessent Shell 2014.4-prerelease)
File Edit Display Data Tools Windows Help
Hierarchy Library Clocks DRC Violations
Instance Name
Signals
Console
BUILD NEW GUI
RUN NEW GUI - TARUN
Shell

```
/bsr_i1/vin_66
/bsr_i1/vin_67
/bsr_i1/bsc_dosync
/bsr_i1/bsc_dosync/ix33
/bsr_i1/bsc_dosync/reg_latch_out
/bsr_i1/bsc_dosync/reg_serial_output
/bsr_i1/bsc_dosync/ix31
/bsr_i1/bsc_v_dout_7
/bsr_i1/bsc_v_dout_7/ix33
/bsr_i1/bsc_v_dout_7/reg_latch_out
/bsr_i1/bsc_v_dout_7/reg_serial_output
/bsr_i1/bsc_v_dout_7/ix31
/bsr_i1/bsc_v_dout_6
/bsr_i1/bsc_v_dout_6/ix33
/bsr_i1/bsc_v_dout_6/reg_latch_out
/bsr_i1/bsc_v_dout_6/reg_serial_output
/bsr_i1/bsc_v_dout_6/ix31
/bsr_i1/bsc_v_dout_5
/bsr_i1/bsc_v_dout_5/ix33
/bsr_i1/bsc_v_dout_5/reg_latch_out
/bsr_i1/bsc_v_dout_5/reg_serial_output

SETUP> //  command: find_design_names

=== No matching netlist instance found

SETUP> //  command: find_design_names

=== Found 4 netlist instances ===
/bsr_i1/bsc_clka
/bsr_i1/bsc_clka/vin_68
/bsr_i1/bsc_clka/reg_serial_output
/bsr_i1/bsc_clka/ix21

SETUP> 
```

Search: /bsr_i1/bsc_clka
Search Results: lka/reg_serial_output\n/bsr_i1/bsc_clka/

Search Options
Search Depth: ● Entire Hierarchy ○ Specified Hierarchy
Search Area: ○ Topmost Library Cells ● All Netlist Instances ○ All Levels Of Library Cells ○ Any Level Of Design or Library Cells
Search For: ● Instances ○ Ports And Nets ○ Pins
Limit Displayed Matches To: 25

Submod: //bsr_i1/bsc_edt_channels_in1

Summary: It has been observed that a fabless designer would like to switch between the main shell and the GUI depending on the comfort level of the underlying process he is working on. GUI becomes really handy when a user has limited exposure to a particular design methodology; however the design process becomes faster when in non-GUI mode given that he understands the subject really well. Under such situations, it is important to give the flexibility to a user to move the between the 2 modes without actually exiting the tool. Further, the GUI should also be given a console in GUI so that he can execute scripts/commands from within the GUI itself along with several other advantages such as hyperlinking etc. that a console can offer to add value to the design process of a fabless design engineer.

We present the pseudo code as under for starting the main console on a main interpreter while registering the DFTVisualizer GUI on a slave one. The approach is extensible and can be adopted easily by anyone who wishes to make such a transition. The following explains the algorithm and other necessary details that are required to accomplish the task of having a TCL based GUI and a TCL based main shell in parallel.

Setting up things for main-shell and DFTVisualizer GUI co-existence

TCL Initialization function doing the requisite setting

```
Tclappinit()
{
   // Calculate platform specific info and set it properly to invoke the GUI

    tcl_lib_relative     = gen_lib_relative + "/tcl" + TCL_VERSION;
    tck_lib_relative      = gen_lib_relative + "/tk" + TK_VERSION;
   // eltclsh make the main console tclish in nature and can accept all tcl tk commands
    eltclsh_lib_relative = gen_lib_relative + "/eltclsh-1.11.1



  // initialise the master interpreter on which the main shell/console shall be registered
  tclInterpreters_[MasterInterpreter] = Tcl_CreateInterp();

  // Override default Tcl pre init script because of the bug during creation of slave interp
  // It will also set tcl_library and tk_library appropriately
  setTclPreInitScript(tclpath, tkpath);

  // initialise the master interpreter
  Tcl_Init(masterInterp())

  // initialise the slave interpreters one of which will have the DFTVisualizer GUI
 // Very important to note here that GUI gets to the slave interp
 for (unsigned i(0); i < TclInterpreterTag_END; ++i) {
    TclInterpreterTag tag(static_cast<TclInterpreterTag>(i));
    if (MasterInterpreter == tag) { continue; }
    if ((DftvInterpreter == tag) && !createDftvInterp()) { continue; }
    tclInterpreters_[tag] = Tcl_CreateSlave(masterInterp(), nameOf(tag).c_str(), 0);
  }

  // initialise eltclsh which forms the backbone for  TCLishing of the main console and set it up properly
  for (unsigned i(0); i < TclInterpreterTag_END; ++i) {
      TclInterpreterTag tag(static_cast<TclInterpreterTag>(i));
      Tcl_Interp* interp(getTclInterp(tag));
      if (!interp) { continue; }
      Tcl_SetVar(interp, "eltclsh_library", eltclshpath.c_str(), TCL_GLOBAL_ONLY);
  }

  // Setup all the interpreters with proper TCL variable values
  for (unsigned i(0); i < TclInterpreterTag_END; ++i) {
    Tcl_ListObjAppendElement(interp, auto_path, Tcl_NewStringObj(genlibpath.c_str(), -1));
    Tcl_ListObjAppendElement(interp, auto_path, Tcl_NewStringObj(tclpath.c_str(), -1));
```

```
     Tcl_ListObjAppendElement(interp, auto_path, Tcl_NewStringObj(tkpath.c_str(), -1));
     ………………
     Tcl_CreateCommand(interp, "TclAppPrompt", TclAppPrompt,
       (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
     Tcl_Eval (interp, "set tcl_prompt1 TclAppPrompt" );
     ……………….
  }

  /* Create a body for unknown and Call Tessent_Unknown from unknown.
   * The reason for this indirection is because some packages (e.g. itcl) use [info body unknown] during runtime.
   *Using UPLEVEL to 1: Make the additional indirection transparent to our unknown handler.
   * 2: Explode the var arg list (args) back to individual arguments before calling our unknown handler.
   */
  // string TessentUnknownBody = "if {[info command ::Tessent_Unknown] != \"\"} {return [uplevel
Tessent_Unknown $args]}";
  string TessentUnknownBody = "if {[info command ::Tessent_Unknown] != \"\"} {\n";
  TessentUnknownBody       += "   set newcmd [linsert $args 0 ::Tessent_Unknown]\n";
  TessentUnknownBody       += "   uplevel 1 [list ::catch $newcmd ::tcl::UnknownResult
::tcl::UnknownOptions]\n";
  ………………….

 // Create a new Tcl command called Tessent_Unknown and bind it to our unknown handler
     Tcl_CreateObjCommand( interp, "Tessent_Unknown", UnknownCmd, (ClientData)NULL, NULL );
     Tcl_Eval(interp, unknownDef.c_str());
  }
}
```

DFTVisualizer GUI interaction with “main shell”/”kernel” for GUI operations

Now, Visualizer GUI has to interact with the main shell in the form of commands and it is done is Visualizer GUI as follows:

```
//call the C registered TCL command from tcl/tk
c_DftKernelExecute “$command”

 // since proper escaping / issuing modern commands, on DFTVis Transcript is the onus of the  user.
string tclcmd(data);;
if(!isCallFromTranscript()) {
    // first convert the command into 3_2_1 syntax
    tclcmd = ConvertToModernCommand(tclcmd);
    // special handling in case there are escape characters in the command
     tclcmd = ComTcl_EscapeForTcl(tclcmd, false, true);
 }

// execute the command in the core command handler
// DFTApp is the class that manages the command execution of
// the main shell
DftApp::get().execCommand(data, msg, !isCallFromTranscript(), !isCallFromTranscript())))
```

ELTCHSH to make main shell TCLish

Meanwhile, we have adopted “eltclsh” (editline tcl shell) an interactive shell for the TCL programming language. It provides command line editing, history browsing as well as variables and command completion thanks to editline features. The completion engine is programmable in a way similar to tcsh, and comes with an intelligent completion for the full TCL language by default. eltclsh is an open-source software released under a BSD license. In order to use eltclsh as the main shell command, following needs to be done:

```
// Setting the variables for eltclsh and calling the initialize function of eltclsh
// which runs a while(1) loop for “smart” input options on main shell
Tcl_SetVar(interp, "eltclsh_library", eltclshpath.c_str(), TCL_GLOBAL_ONLY);
if (Eltclsh_Init(interp) != TCL_OK) {
    Msg::Error("Failed to initialize eltclsh.");
    return TCL_ERROR;
}
```

Please note that the pure eltclsh needs to be tailored to the tool own needs while performing the handshaking between the command dictionary (for example) to do command completion. Following is the routine that we have implemented at our end that helps accomplish the same. It would be good to mention here that all the Tcl_CreateCommand registered commands on a given TCL interpreter are visible to the “eltclsh” as well and is used while performing various key functions such as TAB completion, arrows keys, history etc.

Making ELTCLSH to work with our GUI

```
// Our Custom command-line completion procedure that operate on a 3_2_1 commands
unsigned char elTclNewCompletion(EditLine *el, int ch)
{
  ElTclInterpInfo *iinfo; const LineInfo *linfo;
  // get context on which the commands registered will show up and matched against
  el_get(el, EL_CLIENTDATA, &iinfo); linfo = el_line(el);

  // compute current command line: it is the concatenation of the current command
 // (any incomplete lines entered so far) plus the current editline buffer
  cmd[1] = Tcl_DuplicateObj(iinfo->command);
  cmdLine = Tcl_NewStringObj(linfo->buffer, linfo->cursor - linfo->buffer);
  Tcl_AppendObjToObj(cmd[1], cmdLine);

 // call the procedure that generates completion matches
  sprintf(buffer, "%d", iinfo->windowSize);
  cmd[0] = Tcl_NewStringObj("el::Get_Completion_Data", -1); // this is captured below
  Tcl_IncrRefCount(cmd[0]); Tcl_IncrRefCount(cmd[2]);
  if (Tcl_EvalObjv(iinfo->currentInterp, 3, cmd, TCL_EVAL_GLOBAL) != TCL_OK) {
    printf("\n Command Error: %s\n", Tcl_GetVar(iinfo->currentInterp, "errorInfo", TCL_GLOBAL_ONLY));
    el_beep(el); return CC_REDISPLAY;
  }
  // handles different cases based on number of matches
  // no match
  if (count == 0) return CC_ERROR;
  Tcl_GetIntFromObj(iinfo->currentInterp, matchList[2], &start);
  el_deletestr(el, linfo->cursor - linfo->buffer - start);

// Unique Match
  if (count == 3) {
    el_insertstr(el, Tcl_GetStringFromObj(matchList[1], NULL));
    return CC_REFRESH;
  }

  // Multiple Match
  if (count == 4) {
    Tcl_ListObjGetElements(iinfo->currentInterp, matchList[0], &count, &matches);
     // ask user if matches exceed threshold
    if (count > iinfo->completionQueryItems) {
      printf("\nDisplay all %d possibilit%s? [y/n] ", count, count>1?"ies":"y");
      fflush(stdout);

     // process the information based on “y”/”n” received by the user

    // restore back the text user typed before pressing TAB
    el_insertstr(el, Tcl_GetStringFromObj(matchList[1], NULL
    return CC_REDISPLAY;
   }
  }
     // put the results on the standard output
    fputs(Tcl_GetStringFromObj(matchList[3], NULL), stdout);
  }
}
```

```
// TCL proc that gives data to above function
proc Get_Completion_Data {partialWord windowSize} {
     // Given a partialWord and windowSize, return the following list
     //            output = [list matchList display start formattedDisplay]
    //  Note       : Uses el::matches (of “eltclsh”) to obtain the matches first

    // Step 1: Verify whether we should have TAB completion
    if {!$auto_completion_flag} {return {}}
    // Step 2: Prevent TAB on empty shell
     set regexpr {^[ \t\n]*$}
     set regexpVal 0

    // Use catch, otherwise regexp will crash on some cases (like unmatched paranthesis)
     catch {set regexpVal [regexp $regexpr $partialWord]}
     if {$regexpVal} {return {}}

     // Step 3: Obtain the matchList: Format is [start end matches]
     // this is the function that performs the matches with commands registered with the interpreter
    // and return the options
     set matchList [matches $partialWord 1]
     set returnList [list {} 0 0 0 0]

    // Step 4: Parse the matchList
     set matchListSize [llength $matchList]
     if {$matchListSize == 0} {
       // If there is no match, return NULL
     } elseif {$matchListSize == 1} {
       // If there is unique match, return with appended space
     }

     // Step 5: Find the largest common substring
     set actualList [list]; set formatList [list]; set first true; set intersectString ""
     set intersectString [largestSubString $actualList $formatList]

    // Step 6: Get Formatted display for multicol multirow display
     set formatString [GetFormattedString $formatList $windowSize]
     return [list $actualList $intersectString $startVal $formatString]
   }
}
```

Using “eltclsh” in the application (or GUI) side

“***eltclsh***” is used on the application side (DFTApp in our case) under different scenarios as follows in order to provide the smart command editing functionalities

Case 1: In order to disable it for some commands/commands not following a given syntax: e.g. we want eltclsh to honor only 3_2_1 and not support “3 2 1” syntax we disabled it for latter case.

```
  /* Turn Off TAB Auto-completion for LEGACY “3 2 1“commands */
    if (GlobalCommandDictSingleton::getCommandDict().commandStyle() == CommandDict::LEGACY) {
      int status = Tcl_Eval(interp, "::el::Set_Auto_Completion 0");
      if (status != TCL_OK) {
        Msg::Error("Failed to Disable TAB Auto-Completion. Reason: %s", Tcl_GetStringResult(interp));
```

```
          return TCL_ERROR;
        }
        Tcl_ResetResult(interp);
      }
```

Case 2: Retrieving the history of commands

```
  int status = Tcl_Eval(currentInterp(), "el::get_history_data history_line");
    if (status != TCL_OK) {
      printf ("\nFailed to access History Info. Reason : %s\n", Tcl_GetStringResult(currentInterp()));
    } else {
      const char* line = Tcl_GetStringResult(currentInterp());
      if (line) sscanf(line, "%d", &size);
    }
```

Case3: Adding the command to the eltclsh history

```
   string command = string("el::add_history_data ") + string("{") + line + string("}");
    status = Tcl_Eval(currentInterp(), command.c_str());
    if (status != TCL_OK) {
      printf ("\nFailed to edit History Info. Reason : %s\n", Tcl_GetStringResult(currentInterp()));
    }
```

Case 4: Showing the completion matches on pressing the tab

```
  cmd[0] = Tcl_NewStringObj("el::Get_Completion_Data", -1);
  cmd[1] = Tcl_NewStringObj(partialWord.c_str(), -1);  // such as “se” for “set_system_mode”
  cmd[2] = Tcl_NewStringObj(buffer, -1);

  int status = Tcl_EvalObjv(currentInterp(), 3, cmd, TCL_EVAL_GLOBAL);
  if (status != TCL_OK) {
    printf ("\nUnable to successfully find autocompletion tokens. Reason : %s\n",
Tcl_GetStringResult(currentInterp()));
    Tcl_ResetResult(currentInterp()); return "";
  }

  int count = 0; Tcl_Obj **matchList = 0;
  cmd[0] = Tcl_GetObjResult(currentInterp());
  Tcl_ListObjGetElements(currentInterp(), cmd[0], &count, &matchList);

  /* no match */
  if (count == 0) return "";

  Tcl_GetIntFromObj(currentInterp(), matchList[2], &start);
  display = string(Tcl_GetStringFromObj(matchList[1], NULL));

  if (count == 3) return "";
  if (count == 4) {
    Tcl_ListObjLength(currentInterp(), matchList[0], &matchCount);
    return string(Tcl_GetStringFromObj(matchList[3], NULL));
  }
```

DFTVisualizer GUI transcript made “mirror image” of main shell

Further GUI should have a console that helps to issue commands and have helpful features for the operations in it. Here, the text widget has been enhanced in such way that it mirrors the shell window of the DFT tools. This means that all messaging, commands issued, error messages, which are shown in shell window of the DFT tool are also now seen in this text widget. The text widget also supports tab, history and UP/Down arrow keys. This text widget also displays the line with error messages, commands and the warning messages in a red, black and green color respectively. The errors messages, commands and warning messaged are highlighted with different color to help the user point out the problematic line easily thus helping the user to debug the issue further. Some of relevant text in the widget for e.g. files names, instance names, DRC ID and lines are also displayed as hyperlinks. When the user clicks on these hyperlinks, the relevant information is shown in the other windows of the tool, helping the user to debug the issues related to Design for Test. The following flowchart depicts how the mirroring of the DFTVisualizer console and the main shell has been done in our tool.

To mirror the command execution in console, the command entered, command result and prompt displayed on shell/console, gets also displayed in console/shell and for that three callbacks have been set.

1. PrologueCallback (prologue_callback)
   This callback gets called before the command execution. This callback is set to display the command entered on shell in console and vice versa.
2. CommandResultRedirectCallback
   The callback is called when the command is executed .In this callback the command results are stored in internal buffer
3. EpilogueCallBack(epilogue_callback)
   The callback is called after the command execution and it displays the command prompt that displayed on shell in console and vice versa.

The following pseudo code depicts the mirroring of console to Shell window

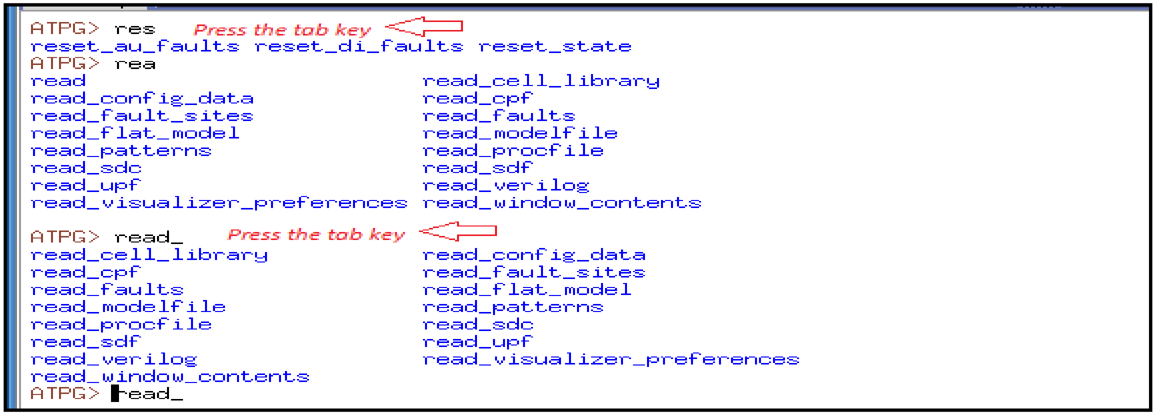
```
// initialize the callbacks
int IntializeCallBacks (  Tcl_Interp *interp) {
     prologue_callback(prologue_callback);
     command_result_redirect_callback(command_result_callback);
     epilogue_callBack(epilogue_callback)
 }
```

Pseudo code for mirroring the command entered on shell

```
        // if the command is issued on Shell then mirroring the same to console

        // this callback is get called before the command execution
         bool prologue_callback string& commandString ) {
             showCommandInConsole(command.c_str());
           }

        // this callback is gets called when the command is executed
        Void  command_result_redirect_callback ( const char* result )
        {
          // add commands result into to the  internal buffer
            appendCommandResultToBuffer(text);
        }

        bool epilogue_callBack (const string& command, int status) {
             display_textFromBuffer_in_console();          // text of internal buffer is displayed in console
             redirect_prompt_from_shell_to_console();   // display the prompt displayed in console
        }

Pseudo code for mirroring the command entered on console
        // this callback is get called before the command execution
        bool prologue_callback string& commandString ) {
              show_command_in_shell(command.c_str());
        }
        // this callback gets called when the command is executed
        void  command_result_redirect_callback ( const char* result )
        {
            display_command_result_in_shell(text);
        }
        bool epilogue_callBack (const string& command, int status) {
                   redirect_prompt_from_console_to_shell();      // display the prompt in shell
        }
```

Mirroring of Tab Support in console

When a tab key is pressed after entering some text on command prompt in shell window, then all the relevant commands starting with the text get displayed in the shell window. The same feature has been implemented in console window. The following snapshot shows the example of tab support in Console.



```
ATPG> res
reset_au_faults reset_di_faults reset_state
ATPG> rea
read                              read_cell_library
read_config_data                  read_cpf
read_fault_sites                  read_faults
read_flat_model                   read_modelfile
read_patterns                     read_procfile
read_sdc                          read_sdf
read_upf                          read_verilog
read_visualizer_preferences       read_window_contents

ATPG> read_
read_cell_library                 read_config_data
read_cpf                          read_fault_sites
read_faults                       read_flat_model
read_modelfile                    read_patterns
read_procfile                     read_sdc
read_sdf                          read_upf
read_verilog                      read_visualizer_preferences
read_window_contents
ATPG> read_
```

Figure : Tab Support In Console.

Flow of Tab support in console

The shell runs on master interpreter and the console on slave interpreter. To have tab key support in console, the results from master interpreted are in stored string and then the same are displayed in the console window.

Pseudo code for tab support in console

```
        // Tcl method called when tab key is pressed after entering some text on console
         Proc  tabExpandMethod {tab_str} {
            set _tabResultString  [Get_Tab_Completion_Data_From_Shell  $tab_str
        foreach tabStr $tabResultString {
          // display each tabStr in proper column format.
        }
    }

    int Get_Tab_Completion_Data_From_Shell   { } {
              elTclParseCommand()
              // format the results in proper columns
               formatCommandResults and return
     }
```

The history command and up/down arrow are also supported in similar way as that of tab support.

Conclusion

In this paper, we have observed the different aspects of co-existence of TCL based interpreter (*eltclsh*) and TCL based GUI (DFTVisualizer) along with a GUI console that mirrors the main shell and how all this was carried out in our tools. Similar approach could be easily adopted in your tools in order to make the non-gui and gui mode coexist.

Bibliography

TCL/TK wiki, http://wiki.tcl.tk
Eltclsh: http://wiki.tcl.tk/11176