Enhanced Tcl/Tk Widgets for EDA Applications

Gaurav Bansal, Roshni Lalwani
DFTVisualizer Group, Mentor Graphics Corporation
{gaurav_bansal | roshni_lalwani}@mentor.com

Abstract
Considering the huge designs required to be navigated and analyzed in the Electronic Design Automation (EDA) applications, there is need to enhance the underlying Tcl/Tk widgets to optimize both the performance and memory consumption.
This paper talks about enhancing the MtiHierarchy widget (part of the Mtiwidget package) and Text widgets in a way that eliminates the performance overheads due to the magnanimity of the data in the EDA applications and also providing various useful features.

1. Introduction

DFTVisualizer is a Tcl/Tk based rich graphical user interface (GUI) which provides a debugging environment to the EDA tools of Mentor Graphics. It is built upon various open-source mega-widgets like MtiWidgets, MtiWaveWidget, Text Widgets.

The biggest challenge in handling huge amount of the data is when it has to be edited in runtime through the GUI. An addition or deletion of a column would take something close to half an hour while handling a typical design which has more than 10M gates. This was a serious limitation for a GUI.

In addition to enhancing the performance of the existing features, supplementary features had to be added in the application to meet the new developments being done in the backend EDA tools. Features like sub-columns and frozen-in-place tree column, did not have an inbuilt support in MtiHierarchy widget.

Similarly it was required to enhance available text widgets to provide a transcript window which is not just a mirror of the command line output, but also provides features like visual differentiation between various types of information as well as hyperlinks so that the user can open the reference documents/files by clicking on the link, start the debugging process by clicking on the errors reported by the tool, etc.

Section 2 below covers MtiHierarchy Widget, highlighting work done to enhance the performance and add new features. Section 3 outlines the enhanced text widgets which support smart hyperlinks.

2. **MtiHierarchy widget**

MtiHierarchy widget is a part of Mtiwidgets, which is a Tcl/Tk based open source package providing various useful capabilities like Hierarchical Tree interface, docbar, toolbar, pane manager, tabbed pane interface, etc.

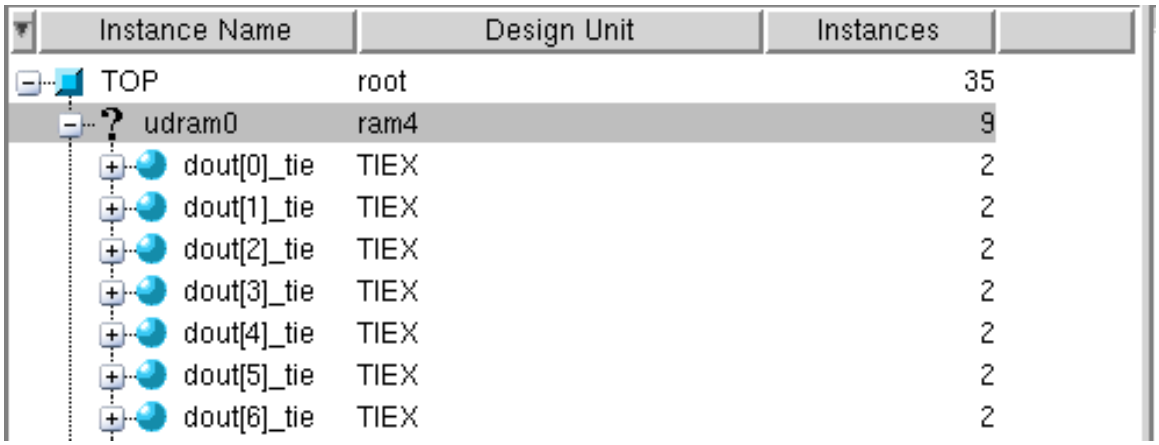

Figure 1 – MtiHierarchy widget

MtiHierarchy widget provides the Hierarchical tree interface as seen in the image.
But with exponential increase in the complexity and size of EDA designs, there was a need to enhance MtiHierarchy widget to avoid associated performance overheads. The approaches for each enhancement along with its limitations and possible solutions to its limitations are discussed further.

a. Callback support to improve the performance

The protocol for displaying hierarchical tree in MtiHierarchy widget involves sending information of each instance (row) from the EDA tool to the widget. Information is passed to the MtiHierarchy widget in the following format.
<name> <app data> <label> <tag> <icon> <label> <tag> <icon>...
The "name" is the unique id of each instance (e.g. memory address) and "app data" is a general purpose field for storing any application specific data. The information to be displayed in each column is sent in a set of three parameters i.e. <label> <tag> <icon>.

The conventional approach to displaying a data in widgets is to pass the information for the entire tree/block irrespective of how many instances are actually displayed in the visible portion of the widget. The widget stores the data for each instance as an "item" (object of data-structure wTreeItem_t) where the data in <name> field is stored in "item->name" for that instance. Data in the <app data> is stored in the "item->appData" and "item->data" stores the sets of <label><tag><icon> of all column corresponding to that instance.

In Figure 1, the data set passed to MtiHierarchy widget for the "TOP" instance is shown in Figure 2.

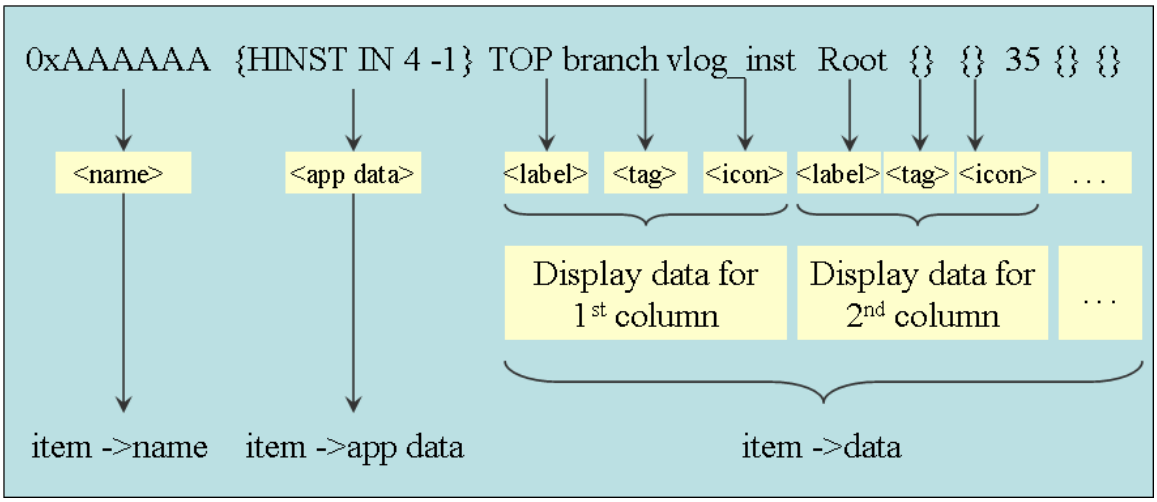

Figure 2 – Protocol for displaying hierarchical tree in MtiHierarchy widget

Widget uses "item->data" for that instance to display the information in each column. Now if the user were to insert a new column in the widget consisting of a tree with 10M instances under top instance. The conventional approach would require iterating over the entire tree and append a set of <label> <tag> <icon> to the list of each instance in the "item->data" field. This is what caused a major performance overhead.

Considering that only about 50 instances are in visible region at a time and the display data is easily retrievable for each instance, given that the widget has instance's <name> and <app data>, an alternative approach is implemented where the information is split into static data (<name> and <app data>) and the dynamic display data (sets of <label> <tag> <icon>) such that the widget stores only the static data and requests the dynamic display data from application as and when required.

For the 50 instances in visible region, the widget requests the display data from the application by passing their <name> and <app data>. So wherever the widget was using the field "item->data" (sets of <label><tag><icon>) to display any instance, now it gets the display data from the application at runtime as:

```
if (treePtr->callback) {
          wtcl_callbackForItemData(treePtr, item);
          list = Tcl_GetObjResult(treePtr->interp);
} else if (item->data){
          list = (Tcl_Obj*) item->data;
}
```

The new approach does not require any modification of the data at the widget side. So addition/deletion of a column does not require any iteration. The new data for the column is provided by the backend application when display data is requested for an instance by the widget.
This approach improved the performance of the application by many folds. Time taken to insert a new column was more than 30 mins for a typical design and with the new approach it takes only fraction of a second.

Limitations and Solutions

1) Sorting: Though this approach is very useful in case of displaying data, but it becomes inefficient when using features which require the entire display dataset to be involved, for example, sorting the tree on a particular column. While sorting with this approach, widget fetches the data from the application every time two instances are to be compared. This adversely affects the performance while sorting.

Sorting on a particular column basically involves sorting the item pointers for each row based on the <label> field of the column. Conventionally, the widget compares the labels of the relevant columns stored in "item->data" and sorts the item pointers. With the new approach we pass the column and the parent instance to the application which returns the list of sorted item->name of its child instances. We have to order the item pointers accordingly.

So we create a hash table with "item->name" as the key and "item" pointer as the value. From the sorted list of item->name returned by the application, we get the corresponding item pointer from the hash table and place them in a sorted array.

With this approach sorting is at least as fast as before for the columns with strings and is faster for columns with numerical values.

Algorithm for sorting

```
if (flag set for backend sorting) {
  entry=Tcl_CreateHashEntry(itemName,item->name,&status);
  Tcl_SetHashValue(entry, item);
}

// Call the application function with parameters like sort
// column name, and parent id and app data, direction of
// sorting, etc

// The application returns the list of sorted item->names

// Iterate over the list and for each item->name get the
```

```
// item from the hash table
if (entry = Tcl_FindHashEntry(hash_itemName, tokens)){
        item = Tcl_GetHashValue(entry);
        // Re-arrange the fields item->parent->firstvis,
        item->siblings, item->nextvis, item->visIdx, etc
        as per the sorted list of items got from the
        above step
}
```

2) Find/Search: With no display data stored in the widget, doing a search for an instance name in the hierarchy tree will again require fetching the display data for each instance and will be more expensive. In order to avoid the overhead of fetching data for each instance from the application we pass the details of the parent while searching at a particular level. At the backend we search for the required pattern in the sorted list associated with the parent and return the relative position (if found) of the matching instance to the widget. The widget calculates the absolute index by adding the parent->visIdx + relative position returned by the application.

b. Sub column support

In EDA applications an instance has some properties which are organized and best viewed as a set of sub-properties. For example we need to classify faults into various categories which have further sub-categories. These can be best displayed as a column of category with sub-categories as sub-columns. From a GUI purpose the view should be such that sub-columns display when a column is expanded in-place to see its sub-categories and collapse it back to see just the category.

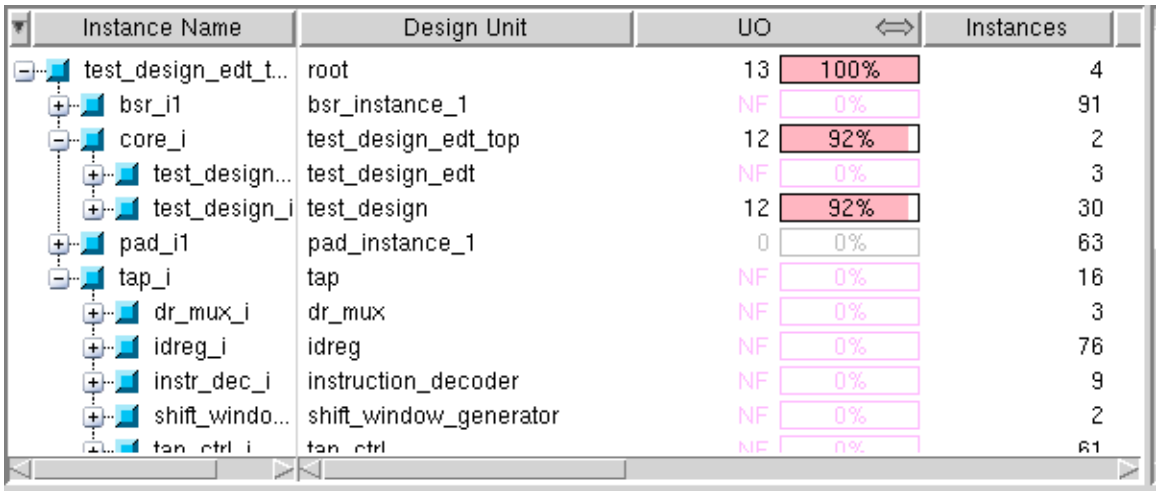

Figure 3 – MtiHierarchy widget with expandable columns

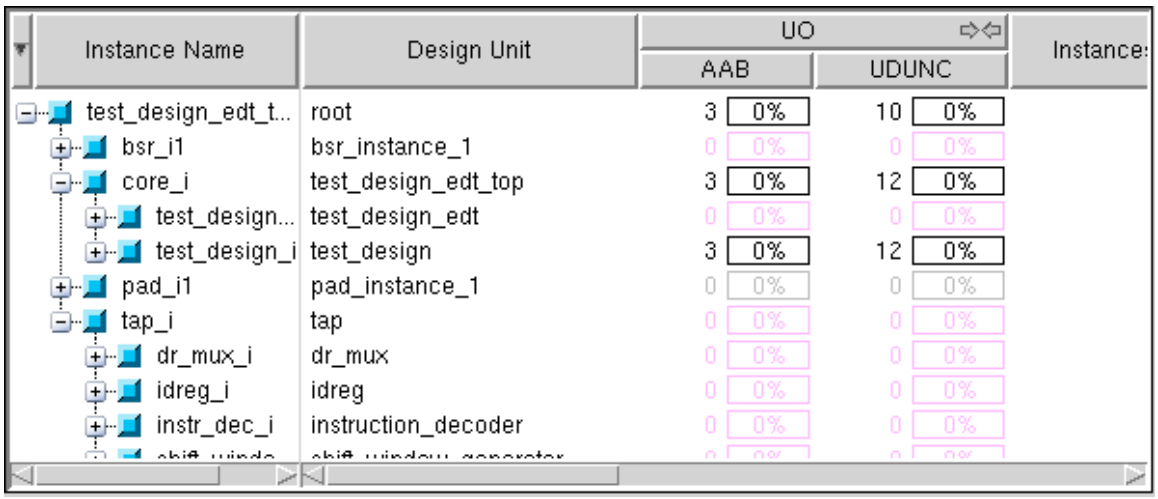

Figure 4 – MtiHierarchy widget with sub-columns

MtiHierarchy widget does not have an underlying support for displaying sub-columns. The available infrastructure in the widget is to create a header button and a column and grid them in the same column. With this available infrastructure there are two approaches that can be followed to display sub-column in widgets.

1) Create sub-columns and pack them inside the column. This will also require that we create a frame for header instead of a header button and pack the header and sub-headers inside the header frame.

Note that the protocol for display data remains the same as before except that the <label> for a column should now be a "list of labels" for a column with sub-columns such that the display data passed to widget for a column with sub-columns will be:

<labe1 label2 label3> <tag> <icon>
where label1 is the label for 1st sub-column, label2 is the label for 2nd sub-column, etc.

Brief overview of the changes required for implementation of sub-columns is as follows:

Provide APIs for sub-column addition, deletion, move, configure, hide, show, cget, index, rename, drawmode, etc. on the same lines as available for columns in class mtiwidgets::Hierarchy in file mtihierarchy.itk.
e.g. we would need to provide APIs for sub-column addition as

```
$_hierWidget column subcolumn add "$column" "$subcol"
$_hierWidget column subcolumn delete "$column" "$subcol"
```

For these operations we would require keeping a map of the name and number of sub-column, just like we do it for columns, and update the variables accordingly.

E.g. For addition of a sub-column we need to keep a map of the number of the new sub-column, its name, parent, etc.
So while adding a sub-column specify the parent column and keep a map of the sub-columns and its parent columns.

```
itk_component add subcol$_subcolNum {
      wvalueWidget  $itk_component($column).$subcolumn \
      itk_component(tree) \
      -parentColumnNum $column_index] ……
#Update the global variables to map the subcolumns-columns
properly
lappend subcol_list $_subcolNum
set _subcol_col_map [lreplace $_subcol_col_map $colIdx
$colIdx $subcol_list]
lappend _subcolParent $col_number

#Generate the corresponding sub headers
subheader add $_subcolNum $col_number
```

To be able to make column expandable/ collapsible we need to provide a clickable icon on the header button. This will require making changes to class mtiwidgets::Colbutton in the file mticolbutton.itk.
We need to define two options: 1) –togglestate to specify the state of the expandable column so that the icon can be updated accordingly. 2) –togglecommand to specify the function to be executed when the icon is clicked.

```
::itcl::body mtiwidgets::Colbutton::_toggleMode {} {
     if {$itk_option(-togglecommand) != ""} {
          uplevel "#0" $itk_option(-togglecommand)
     }
}
```

Function associated with –togglecommand is defined by and in the class mtiwidgets::Hierarchy in the file mtihierarchy.itk as.

```
$hdr configure -togglecommand [::itcl::code $this
_notifyCol_expand_collapse $col]
```

The function in widgets further calls the function in application so that the application can add/delete the sub-column as required.

```
::itcl::body
::mtiwidgets::Hierarchy::_notifyCol_expand_collapse {col} {
     # Notify application about expand/collapse
}
```

This approach has the limitations that since the sub-headers and sub-columns are in different frames/grid cells, there will be algorithmic approach to make their widths align with each other. So whenever we map the sub-columns we have to explicitly make the width of sub-column equal to that of sub-headers.

```
$subcolumn configure -width [winfo width $subheader]
```

Also, since we pack the sub-headers and sub-columns, the user does not have a fine control over the widths of individual sub-column. So when we increase/decrease the width of a column, all the sub-columns increase/decrease proportionally.

2) In order to address the limitations of the previous approach, alternative approach can be considered. Create a placeholder row below the headers to hold sub-headers, if required. When sub-columns are created, create a frame to occupy the 2 rows which will contain the sub-headers and the sub-columns. Grid the sub-headers in row 0 and sub-columns in row 1 of this frame. This will provide the user a lot more control over the sub-column as compared to the previous approach. The only thing that we need to take care is that the headers of the other columns make their row span as 2 so that the headers are the same height as occupied by the header and sub-header combined.

c. Making tree column frozen in place (non-scrollable)

In the current implementation of widget, all the columns are scrolled horizontally with the scrollbar. Since the tree column is the most important column and all the data in the rest of the columns is associated to the instance in the tree column, keeping the tree column frozen while scrolling would provide the user a much better interface. Also, since the design names in many designs are very long, it provides a very convenient way to be able to see the full names of the instances without having to increase the width of tree column.

Usually the horizontal scrollbars are associated with changing the xview of the canvas. So to provide the scrolling to rest of the column, we would require two canvases, one that contains the tree column and the other that contains the rest of the columns. The two canvases will have their own scrollbars that will alter their xviews. So when we scroll the 2nd canvas, only the columns inside that canvas scroll, keeping the tree column frozen. The scrollbar associated with the canvas containing the tree column can be used to scroll the instance names in the tree horizontally.

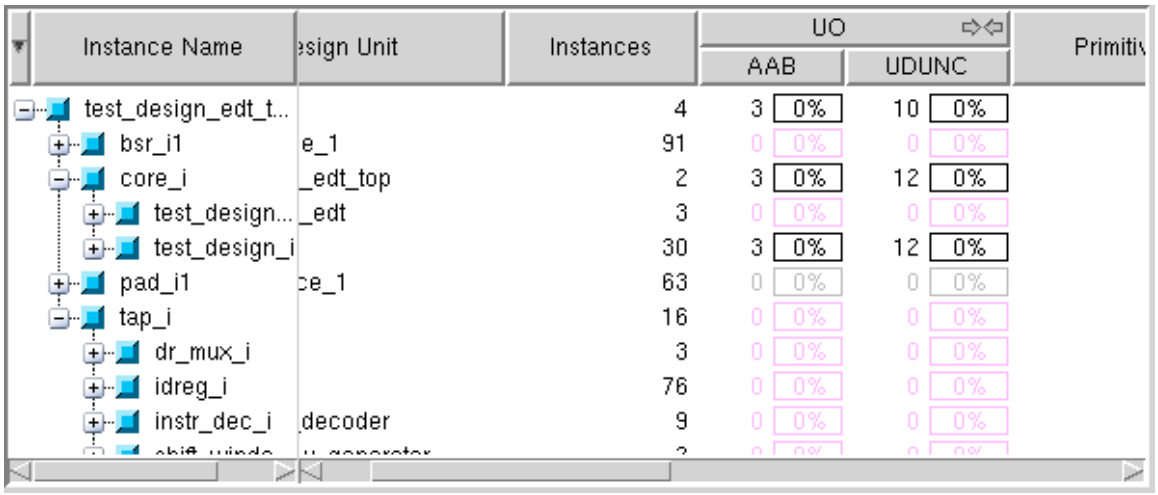

Figure 5 – MtiHierarchy widget with tree column frozen in place

Brief overview of the changes required for implementation of this feature is as follows.

The components of MtiHierarchy widget are in the following hierarchical order:
Clipper (a frame) → Canvas → Sfchildsite (a frame) → Columns

So we create two canvases and grid them inside the clipper as column 0 and column 1 of row 0. We place the scrollbars for both canvases in column 0 and column 1 of row 1 under them.

We configure the 1st scrollbar to alter the xview of the column 0 such that it scrolls the instance names of the tree column and not the column itself. 2nd scrollbar is configured to scroll the xview of the 2nd canvas so that the rest of the columns scroll past the tree column.

```
$sb1 configure -command [::itcl::code $column0 xview]
$sb2 configure -command [::itcl::code $canvas2 xview]
```

One thing that we need to take care of is that now the index of column grid in the 2nd canvas is two less then the earlier index, as the columnspan of the tree column is 2. So if we grid the first column in 2nd canvas, it was the 3rd column in the earlier configuration.

One of the issues faced during this enhancement was that since the header button of the tree column and rest of the column are in different grid systems, it was required to algorithmically align the heights of the headers. So if we have expanded a column in the canvas 2 such that it has sub-column, it requires that we increase the height of tree header so that headers align themselves.

```
itcl::body ::mtiwidgets::Hierarchy::_headerButtonResize
{col} {     ……
     #Get the height of header 1

     # If it has sub-columns then the effective height we
     should consider is 2X

     # Check if there is any height difference in the
     # header rows

     # If tree header is smaller then increase the height
     # of its grid row by extra padding

     # If the tree header is bigger then decrease the
     # padding
}
```

3. Enhanced Text Widget

The current implementation of text widgets is useful for providing an integrated transcript window that 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 can also be seen in transcript window with the underlying text widget. The Enhanced text widget highlights the error messages, commands and the warning messages in a red, black and green color respectively. This text highlighting is to help the user to point out the problematic line easily to debug the issue related to DFT.

Some of relevant text like files names, instance names, Design Rule Violation ID and lines numbers are also made available as hyperlinks. So the user can clicks on the hyperlinks to get to the relevant file or to the relevant information in other windows of the tool to debug the issues very quickly. The snapshot with the transcript window with above features is shown in Figure 6.

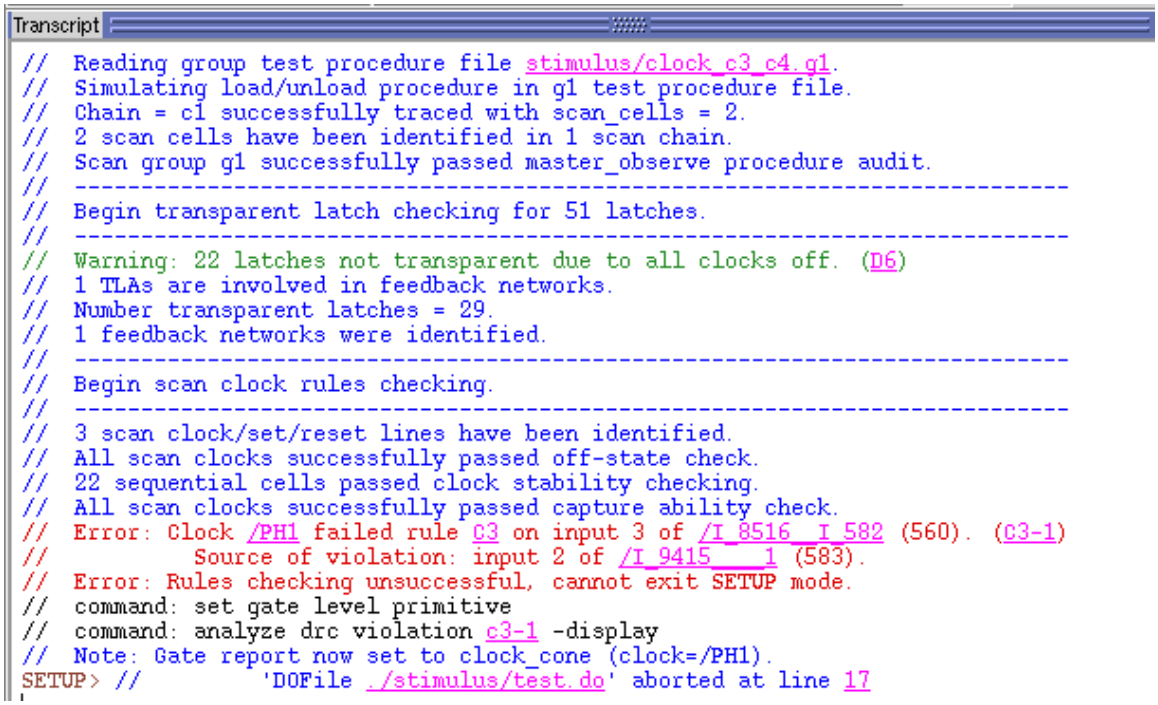

Figure 6 – Enhanced text widgets with hyperlinks

To highlight the (Error/warning/commands) message lines with different color the text in widget has to be parsed line by line to classify the same as error messages, warning messages and commands. To display file names, instance names and Design Rule Violation IDs as hyperlinks, the text in the widget has to be parsed word by word to mark them as hyperlinks.

One of the limitations of providing this enhancement was that, for EDA applications, the size of the log files could be very large. Parsing a big file had performance overheads which made the GUI busy for quite some time. The solution was to provide incremental parsing of the file such that only the portion of the output in the visible region of the widget was parsed.

Algorithm for incremental parsing

1. Load the plan text from file in to the text widget (read & write in chuck size of 256k). This will done only for first time
2. Calculate the boundary lines (B1, B2) of visible area of text widget. Start line = B1 and last line = B2. The algorithm for boundary calculation is shown below.
3. If (text between { B1,B2} is not already parsed) {
   for (line = B1 line <= B2 line++) {
   If(line starts with pattern (Error/Warning/..)

```
            Highlight the line in (red/green/..) color.
            Parse the same the line word by word
         If (word is a filename/Instance name/Design
              Rule Violation ID)
            Mark it as a hyperlink
      }
   }
```

Algorithm for boundary calculation

The boundary calculation is done based on the visible area of the text. If the text between the visible area is already parsed then the boundary calculation method returns null else returns the boundaries.

1. set {frac1,frac2} [$widget yview]
2. set N total number of lines in the text widget
3. set start = N * frac1
4. set end   = N * frac2
5. Set buffer = 10
6. set start = start - buffer
7. set end = end + buffer
8. if {start < 1.0 }
   start = 1.0
9. If { end > last index }
   end = last index
10. foreach {pS pE} [$w tag ranges tagParsed] {
    if { [$pS <= $start] && [$pE >= $start] }
    set start $pE

    if { [$pS <= $end] && [$pE >= $end] }
    set end $pS

    if { [$start >= $end] }
    return NULL

    }
11. $widget tag add tagParsed $start $end
12. Return $start,$end

4. Bibliography

TCL wiki, http://wiki.tcl.tk