Using [incr Tcl] to improve stability of a GUI – A Case Study

Ruchir Agarwal, Prashant Thakre, Manu Goel, Maneesh Agarwal
Mentor Graphics Corporation
{ragarwal | pthakre | mgoel | magarwal}@mentor.com

1.0 Abstract

This paper discusses how a GUI required handling multiple datasets at the same time and providing the user with design and debug information for all the datasets currently open, was having problems in switching between different datasets and different views of a same dataset, due to it being written primarily in Tcl. The paper further describes how the GUI is modified using [incr Tcl] to solve these problems by benefiting from the object oriented concepts built in [incr Tcl]. This paper describes how [incr Tcl] is used by ways of code examples. The paper finally concludes mentioning the benefits achieved by this exercise and also provides recommendations on how to avoid the pitfalls in using Tcl and how they can be easily avoided by using [incr Tcl].

2.0 Glossary

Description of terms used in the paper:

Dataset – database having information about waveform and signal connectivity and design hierarchy for two views, i.e. RTL and Netlist
Restricted Dataset – databases having information about design hierarchy and waveform only. These datasets do not have information about signal connectivity and may support only one view
Wave viewer – widget to view signal waveforms
Pathbrowser – widget to trace signal connections (fanin, fanout) in a design
Hierarchy browser – widget to view design hierarchy of a dataset.
Signal browser – widget to view signals in the selected instance in the *Hierarchy browser*

3.0 Motivation and Problem statement

The primary use of GUI being discussed is to debug designs in post-process mode. Typical use mode is to compile the design, run emulation and generate debug database for waveforms and design connectivity. Each such generated debug database is called a *dataset.* In case of an error, a verification engineer can open a dataset and view the design hierarchy in *hierarchy browser* and *signal browser,* debug by viewing waveforms in *wave viewer,* and browse the connectivity in *pathbrowser*. The requirements from the GUI are:

1. provide ability to debug more than one datasets at the same time
2. *pathbrowser* is locked to a dataset and should allow to trace even if its dataset is not the currently active dataset
3. *wave viewer* should be able to display waveforms for signals from different datasets at the same time
4. *hierarchy browser* and *signal browser* are locked to the currently selected dataset and should switch the design information whenever the dataset is switched or the dataset view is changed.
5. when a dataset is closed, close all the open widgets attached to it and remove signals from the *wave viewer* without closing it as it is a shared window

Unless otherwise stated, the discussion in this section refers to older implementation and specific problems associated with it. The GUI under discussion was written in Tcl and employs global Tcl associative arrays to store information about the active dataset. This global array was used by the widgets to get information regarding the active dataset. When the active dataset was changed, this global array was modified to reflect the change. The global array had element for each variable required for a dataset. So whenever a new feature or task was to be added for the datasets then a new element was added in the global array. With a substantial feature list, this strategy resulted in a lot of such array elements, which were not easy to search in the code and as a result were missed for reset/delete whenever the dataset was closed. This problem was aggravated when the support for *restricted datasets* was required. For these datasets certain array element were not created, so it was also required to check for the dataset type before accessing such array elements. In some cases functions were renamed to provide a constant interface to the display widgets. This made understanding the code flow tough. It was realized with each new task being added, the GUI became more difficult to manage and started impacting the overall stability and quality.

Considering the growing requirements of new features, and need to maintain a very robust intelligent GUI which can handle multiple parallel data-streams seamlessly, it was decided to rewrite a major portion of the GUI. This required encapsulating the implementation in order to force the use of interface corresponding to each type of dataset. We decided to use object oriented programming concepts which lead us to [incr Tcl].

The following sections will describe the GUI architecture and how the old implementation was having problems and how the new implementation solved these problems.

As a summary conclusion, the paper also describes the lessons learnt during this exercise and provides our recommendations to Tcl programmers on how to avoid the problems that we faced.

4.0 GUI Architecture

The GUI comprises of display widget and dataset sources at the backend (Figure 1). Each dataset source has two *view sources* (or display data formats) – RTL and Netlist.

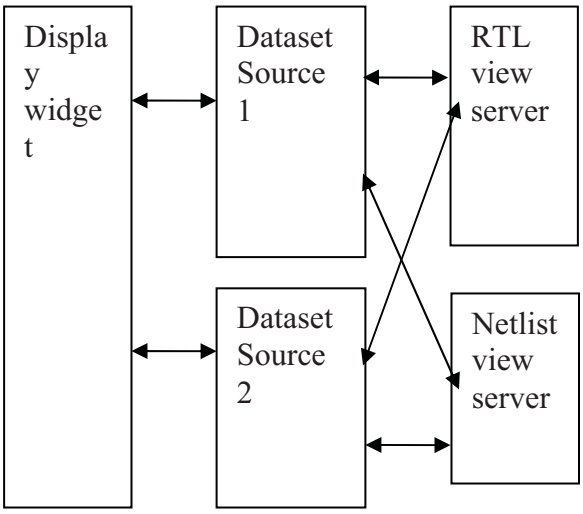

Figure 1: GUI Architecture

For each *dataset* loaded in the GUI there is one unique dataset source with which the display widget communicates. The display widget does not know about the *view sources.* The communication between the

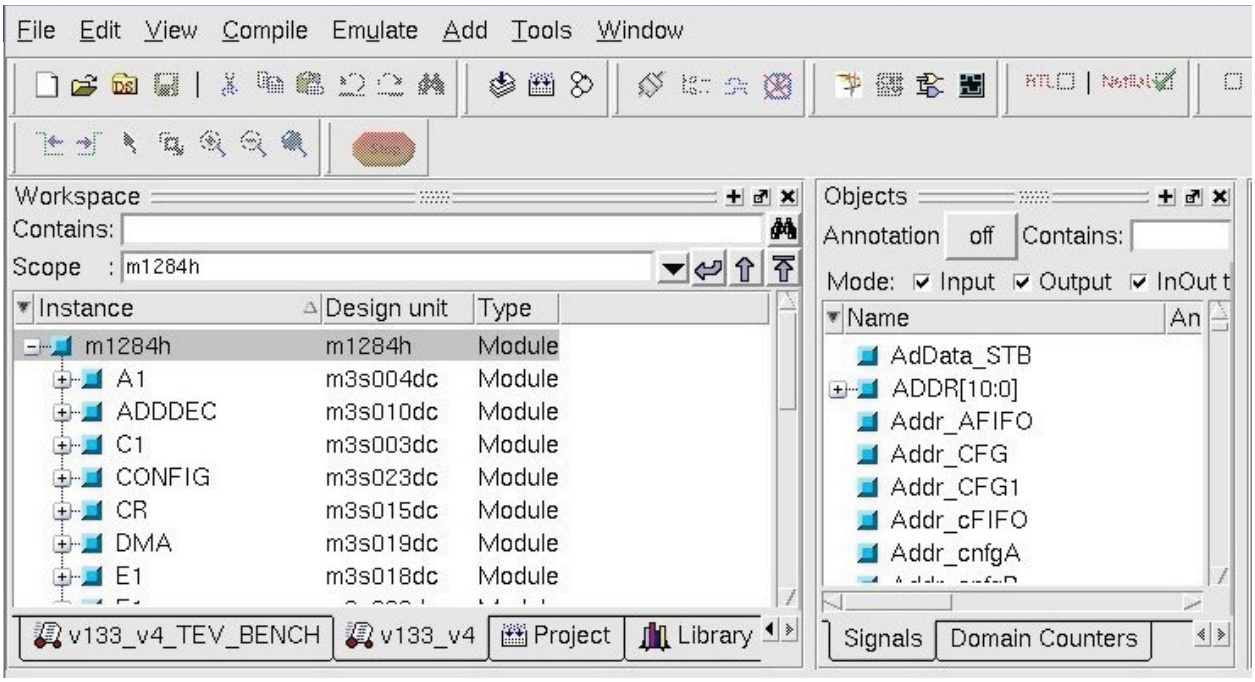

Figure 2: GUI showing multiple datasets

display widgets and *view sources* is routed through the dataset sources.

As seen in figure 2, in Workspace pane, there are various tabs, like v133_v4, v133_v4_TEV_BENCH, etc. Each of these tabs corresponds to a *dataset.* In the figure the *hierarchy browser* for dataset *v133_v4* is being shown and the *Signals* tab in the Objects pane is showing the signals for the currently selected instance. The design hierarchy has two views, RTL and Netlist. In Figure 2, Netlist view is selected. Depending on the view selected – RTL or Netlist, the *dataset source* sends a request to the respective server to get the data to be displayed.

To view the connectivity information the display widgets like *pathbrowser* and *Wave viewer* also communicate with the view server through the dataset source.

4.1 Old implementation

This section discusses the old implementation of the GUI and also mentions the problems seen and remarks on how these problems could be solved by using [incrTcl].
In the old implementation information about various properties of a *dataset* were kept in a global array. Let us call this global array, `g_arr`. So for a *dataset* `ds1`, and property `prop1`, `g_arr` would have a member `g_arr(ds1,prop1)`. All property variables were available in the global scope and could be created, deleted, read and written to from any where. There was no encapsulation. Since the dataset property variables were available in the global scope, it gave us license to create and delete them on the fly. These variables were often created on the fly with an expression of the form:

```
set g_arr($n,prop1) prop_val
```

Notice that the dataset name in the above variable is referred by a variable `$n`. Using variables which were created on the fly and had dynamic names posed the following problems which affected product quality. (1) It was difficult to know if a property variable had been created or not. So a lot of "`info exists`" checks in the code had to be added. (2) Searching of variable created for a dataset in the code was difficult. The only way to know was at runtime using `array names`. This resulted in duplication of information, which was difficult to track and was dangerous because there was a high chance that during dataset switch some of the array members were not updated. (3) When a *dataset* was closed, property variables which were created on the fly were missed out on deletion. To solve these problems coding guidelines could be strictly enforced, but there was no way that such protections could be in-built in the system.

Since the *view servers* provided almost identical information, so the access functions had identical signatures. Depending on the active view, the corresponding access function was called from within the *dataset source*. To avoid `if-else` checks at various places in the *dataset sources* we clubbed all the access functions into one for each view and then used the concept of C function pointers. So depending on the view the functions were renamed. The rename and re-rename used to happen at every view change. So at any point of time it was difficult to know what the active function was. Debugging was tough. A case for polymorphism.

For *restricted datasets*, some of the properties were not available, so before accessing any property we had to add either a check for the dataset type or `info exists` on the property variable. As only one view was available for *restricted datasets* the functioning renaming and re-renaming was also put under a dataset type check. This is a case for inheritance.

4.2 New implementation

We understood the weaknesses of the old implementation and remedial action as underlined in section 4.1 and resolved these in the new implementation.

An abstract base class `dataset_base` was created for common dataset properties. The implementation of *restricted dataset* and *dataset* classes handled the differences by using the concept of inheritance.

Code snippet for the `dataset_base` class:

```
itcl::class dataset_base {
  ## logical name for the
dataset
  protected variable d_ds_name
""

  ## path of the dataset
  protected variable d_ds_path
""

  protected variable d_win_list
[list]

  ## Restrict object creation
  constructor {} {
    # Simulate abstract base
class
    if {[namespace tail [info
class]] eq "dataset_base"} {
      error "Error: can't create
dataset_base objects - abstract
class."
    }
…
```

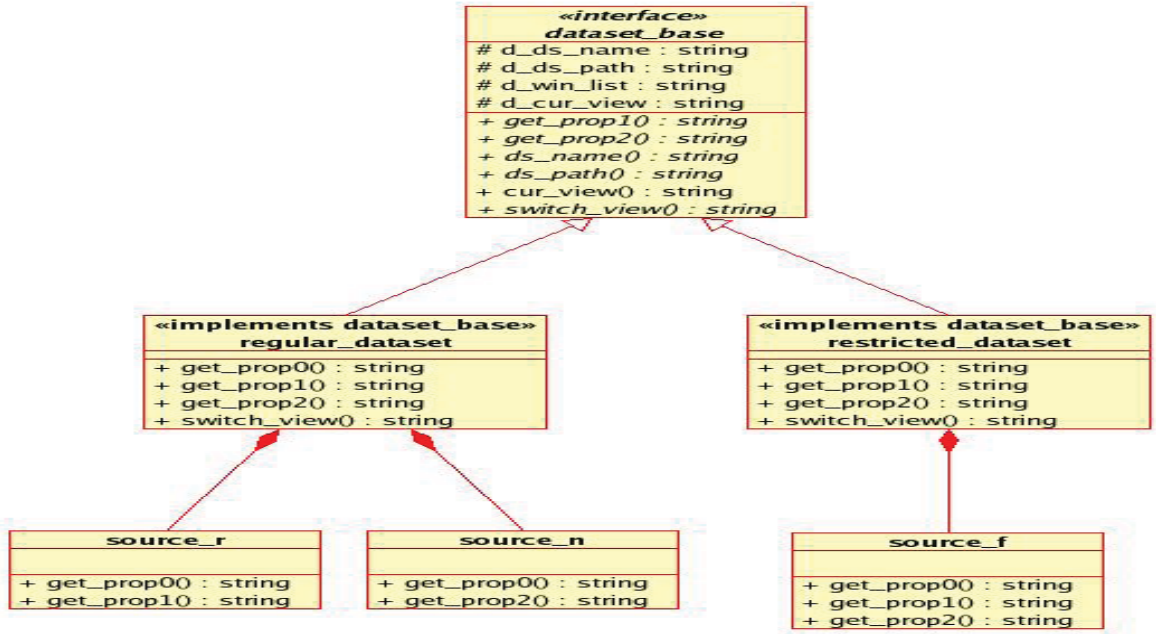

Figure 3: Class diagram

```
…

  public method ds_name {} {
    return $d_ds_name
  }
  public method ds_path {} {
    return $d_ds_path
  }
  public method set_ds_name
{name} {
    set d_ds_name $name
  }
  public method set_ds_path
{path} {
    set d_ds_path $path
  }

…
…

  # empty functions which will
be overwritten in the derived
class

  public method add_win {win} {
    lappend d_win_list $win
  }

  public method delete_win {win}
{
   # search and remove the
window from the list
  …
  …
  }
  public method switch_view {view}
{
  }
  public method get_prop1 {} {
    return “”
  }
  public method set_prop1 {val} {
  }
  public method get_prop2 {} {
    return “”
  }
  public method set_prop2{val} {
  }
...
...

}
```

Code snippet of the `dataset` class:

```
itcl::class dataset {
```

```
  inherit dataset_base

  private variable d_cur_view
  private variable d_prop1
…
…

  public method cur_view {} {
    return $d_cur_view
  }
  public method get_prop1 {} {
    return $d_prop1
  }
  public method set_prop1{val} {
    set d_prop1 val
  }


  public method switch_view {view}
{
    set d_cur_view $view
  }

…
…

}
```

Code snippet of the restricted dataset class:

```
itcl::class restricted_dataset {
  inherit dataset_base

  private variable d_prop2

  public method cur_view {} {
    return "rtl"
  }

  public method get_prop2 {} {
    return $d_prop2
  }
  public method set_prop2 {val} {
    set d_prop2 val
  }

…
…

}
```

As can be seen in the code snippets, common properties - `d_ds_name`, `d_ds_path` and there set and get functions are encapsulated in the base class `dataset_base`. Dummy implementation of dataset specific property functions are also provided in the `dataset_base` class. These functions are overridden depending on the need in the derived classes. For example, the get and set functions for property `d_prop1` are overridden in `dataset` class and the get and set functions for property `d_prop2` are overridden in `restricted_dataset` class.
Functions which are supposed to return a specific value depending on the dataset type, like the function for current dataset view `cur_view`, have different definitions in the respective classes.

As can be seen in these code examples the new object oriented architecture provided us a way to encapsulate all the properties of the datasets in the dataset classes. This helped in setting all the property variables when a dataset was loaded in the dataset object's constructor. And the cleanup was also easy as it could be easily managed in the destructor of the dataset object. The polymorphism and inheritance feature ensured that tricky issues like function renaming were automatically taken care off.

5.0 Results

The new GUI makes successful use of concepts of encapsulation, inheritance, and polymorphism with help of [incr TCL]. The new architecture has also reduced the code base by around 15% compared to earlier implementation. The new re-architected GUI is very stable and robust, and that is

also evident from field and customer usage. The incoming bug rate has seen a reduction of 50% every month, over last few months, despite the concern that major re-architecture could introduce new issues. There are no open issues which are critical in nature

6.0 Lessons learnt and recommendations

1. Do not use variables with dynamic names created in the global namespace. Searching these variables is very difficult.
2. Renaming functions back and forth is difficult to maintain and debug.
3. Segregate data display logic and data. Keep the GUI code thin.
4. Contextual menus should be activated-deactivated by the objects on which these work instead of having global variables which are set/reset depending on the active objects.
5. If windows are attached to different datasets, then it is better to have the list of windows with the dataset object instead of a global list of all open windows. This helps in closing the windows along with the closing/destruction of the corresponding dataset.
6. Code should not rely on state variables

7.0 Bibliography

1. TCL wiki, http://wiki.tcl.tk
2.[incr Widgets] An Object Oriented Mega-Widget Set, Mark L. Ulferts, http://incrtcl.sourceforge.net/iwidgets/paper/paper.html