Too Many Windows

Ron Wold
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070
503-685-0878

Abstract

Over the past 10 years the Modelsim GUI, written in Tcl/Tk, has grown from a simple user interface with three panes to an elaborate interface comprised of over 50 distinct windows. The Modelsim GUI architecture, while sufficient at the time of creation, began to crumble under the weight of so many windows. This paper will explore the issues that occurred as the window count grew and describe the approach taken to resolve these issues.

Keywords

Tcl/Tk, GUI widgets, window layout, pane widget

1. Introduction

Modelsim is an integrated development environment (IDE) used by electronic designers to develop, debug, simulate and test electronic designs. It supports several different hardware description languages (HDLs) - such as VHDL [1] and Verilog [2] – each of which employs unique concepts that require unique user interfaces. As HDL conceptual capabilities expanded over the years, the number of unique windows within Modelsim multiplied to accommodate those new capabilities. Modelsim currently contains over 50 different windows. This large number of windows impacts both internal tool development and customer usage.

2. Menu System & Command Routing

Consider the following simple Tcl/Tk example application, which has a single window (figure 1). A window developer is tasked with implementing a delete command for this window. The command will originate from the pull down menu item **Edit->Delete**.

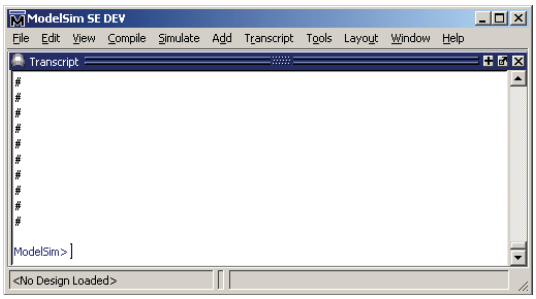


Figure 1

This feature is simple enough, assuming the menu bar object is "`.menubar`" and the pulldown menu **Edit** has already been defined, the following line will implement the menu item.

```
.menubar.edit add command -label "delete" -command { transcript_delete_cmd }
```

The developer must also define the procedure "`transcript_delete_cmd`", which performs the actual deletion. Next, the developer wishes to enhance the menu item such that it is enabled or disabled based on the state of the

transcript window. If there is text selected within the transcript window, the delete menu item should be enabled. If nothing is selected, the menu item should be disabled.

Let's assume that a procedure called "`transcript_something_is_selected`" exists and that it returns true if there is text selected. The menu item state can be set using the menu's post command, which is implemented with the following code.

```
.menubar.view config -postcommand [code view_postcmd]

proc view_postcmd { } {
        If { [transcript_something_is_selected] }{
                $menu.view entryconfigure "Delete" -state normal
        } else {
                $menu.view entryconfigure "Delete" -state disable
        }
}
```

Next, let's add a second window to the application (Figure 2).

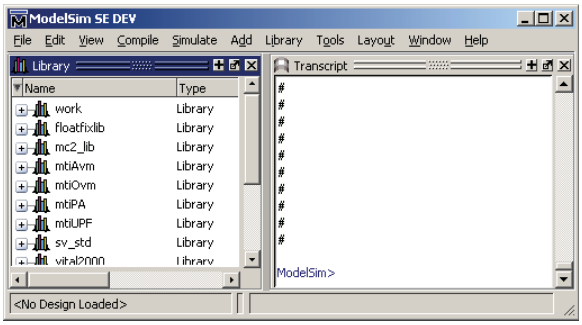


Figure 2

Assume that both windows must support the delete command and both windows must enable or disable the delete menu item based on their selection state. Also assume that we can determine which window is active by calling the procedure `ActiveWindow`. This routine will return which window is active and with two windows the possible values are Library or Transcript.

Using the example above, there are two changes necessary to support the addition of the second window. Both the post command and the delete command must be modified to detect which window is active and to make the appropriate call.

```
proc view_postcmd { } {
        if { [ActiveWindow] eq "transcript" } {
                # transcript is active
                If { [transcript_something_is_selected] }{
                        $menu.view entryconfigure "Delete" -state normal
                } else {
                        $menu.view entryconfigure "Delete" -state disable
                }
        } else {
                # Library is active
                If { [library_something_is_selected] }{
                        $menu.view entryconfigure "Delete" -state normal
                } else {
                        $menu.view entryconfigure "Delete" -state disable
                }
        }
}
```

```
proc delete_cmd {} {
        if { [ActiveWindow] eq "transcript" } {
                # transcript is active
                Transcript_delete_cmd
        } else {
                # Library is active
                Library_delete_cmd
        }
}
```

As you can see, adding a second window adds complexity to the code.

This example represents just one menu item in a system with two windows. But consider a tool with 50 different types of windows and multiple pull down menus, each containing several menu items. Thousands of lines of code are required simply to enable and disable pull down menus.

Modelsim used this basic architecture for several years; but as the window count grew, the code became unruly, was prone to bugs, and the effort necessary to implement a new window became increasingly time consuming.

This issue is not unique to the menus. Key bindings and toolbar buttons have identical problems. In fact, any command that is directed at the active window has this issue. The fundamental problem is that code that can initiate a command needs the following information:

1 – Does the active window support the command?
2 –Is the active window's state capable of executing the command at this time?
3- What procedure must be called to execute the command in the active window?

Coding a command such that it asks these questions for each window is sufficient when there are only a few windows, but is a poor technique when the window count grows. A solution to this issue was found by implementing a mechanism where the questions above could be answered without having specific knowledge of the active window. The mechanism is much like a traffic cop; it directs traffic (commands) to various locations (windows). Hence, we labeled the mechanism Vcop.

Vcop provides general command routing and it is based upon an active window model. At any given time there is one and only one active window. A user can activate a window by clicking in it. If a user activates a window, the previously active window becomes deactivated.

Vcop implements two key pieces of functionality.

1. Given a command, Vcop can ask the active window if the command is supported.
2. If the command is supported, Vcop can ask the active window to execute the command.

This architecture is implemented through the registration of a window callback. The callback performs different types of actions and has been coined an *action table*. Each window registers their action table with Vcop, and the action table must be in the standard form:

```
proc an_action_table { window operation args } { … }
```

The action table is essentially a switch statement that defines two cases for each command that it supports. Using the Delete example above, a window which supports the delete operation would have these two entries within its action table:

```
        can_delete {
                                If { [transcript_something_is_selected] } {
                                        Return 1
                                }
```

```
                        Return 0
                }

        delete  {
                        Transcript_delete_cmd
                }
```

The "can_delete" determines whether the delete command is currently supported. This case can be used by menu items and toolbars to determine whether they should be enabled or disabled. The second case, "delete" is the actual implementation of the delete command.

The implementation of Vcop and the action table allows a command to be implemented without specific knowledge of a window. Code that can initiate a command asks Vcop whether it is currently a valid operation. If the command is to be executed, it asks Vcop to direct the request to the active window. Developing a new window no longer requires knowledge of, or changes to, all the locations that can initiate a command.

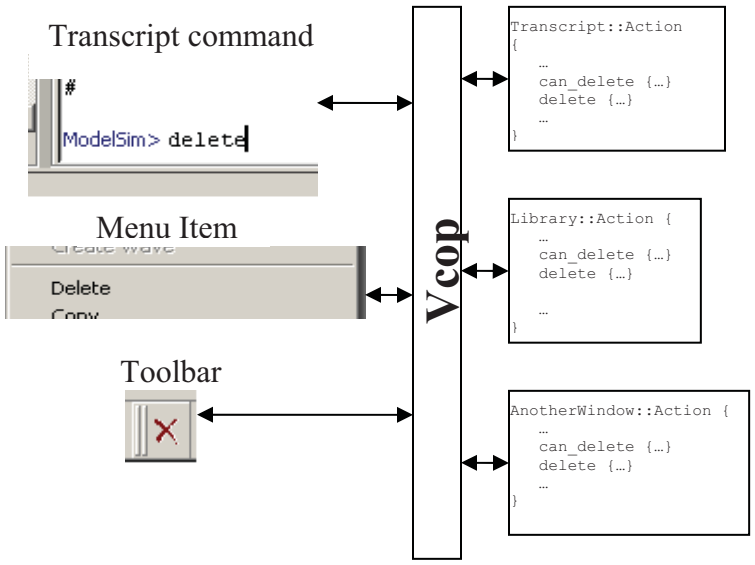


Figure 3

In addition to these changes, a new menu system was defined providing a layer above the intrinsic Tk menu creation commands. This layer has knowledge of Vcop, and manages all calls to Vcop for enabling and disabling menu items. Under the new system, a typical menu item is described as:

```
AddMenuItem "Delete" $edit_menu delete can_delete
```

The argument "delete" is the action entry that should be invoked should the user select this menu item. The argument "can_delete" is the action entry that should be called to determine if the menu item should be enabled or disabled. The menu system controls the menu item states via a menu's post command. When the menu is raised, all menu items that supplied a "can" action are forwarded to Vcop. Vcop sends the request to the active window and the results are returned to the menu system which enables or disables the menu items. If a window's action table does not define a can operation, Vcop considers the command to be unsupported for the window and the item is disabled.

Incorporating Vcop and action tables greatly simplified the menu and toolbar code. The architecture has also reduced the knowledge and effort required to create menu items and toolbars.

An additional benefit of this architecture is the ease with which 3rd party windows are incorporated. Several external groups develop windows that are "sourced" into Modelsim at run time. Prior to these changes, it was virtually impossible for a 3rd party window to tie in the existing menus and toolbars. Under the new architecture, a 3rd party window simply defines an action table and registers it with Vcop.

2.0 Menu Items

Anyone who has grown familiar with a tool, then upgraded to a newer version of the tool and discovered that many of the menu item names or locations have changed understands the importance of menu item stability. Users learn where menu items are in an application. If a menu item's name or location changes, the user is forced to relearn this information.

> "The content of an application's menubar menus should be stable", "…commands in menubar menus should not be present or absent depending on the application's state. To reduce menu complexity, deactivate (i.e., gray out) inapplicable commands rather than remove them"[4].

Two key points made here are: a) the menu picks should not change from release to release, and b) all menu picks should be visible, even if they are not applicable to the active window. In addition to menu stability, menus should not grow too large. A study by Miller found that "eight item menus with a depth of two levels resulted in the fewest errors and fastest retrieval of a designated target" [5].

Historically, when a new window was added to Modelsim, a developer would associate the window's commands to existing menu items that matched. If one could not be found, a new menu item was added. As the window count grew, Modelsim's menus grew, violating the rule on menu size. Rebuilding the menu to display only relevant menu items reduced the overall menu size, but this approach violated the menu stability rule.

Resolution of Modelsim's menu problems involved multiple tasks.

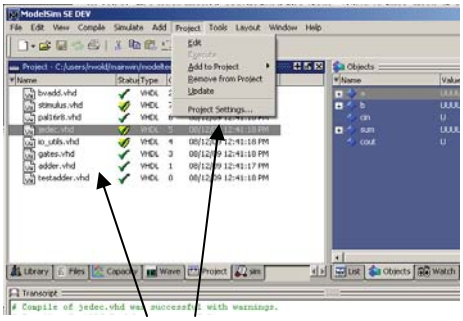
Categorizing Commands – It was necessary to change menu terminology so that a single menu item could be shared by all windows. Changing the menu item terminology violated the rule on menu stability. But if done properly the menu items would not change in future releases as more windows were added.

The top level menu names - such as File, Edit and View - define a category of commands. It was critical that these top level menu names were carefully thought out. They had to be general enough to support all of the current and future menu items. For example, many of Modelsim's windows used the term "insert", while others used the term "include" or "place". We changed the terminology to "add" and incorporated all associated menu items under this main menu label. Eliminating menu names that were similar not only reduced the number of menu items, but also reduced menu search time. For example, if a user wished to "add" something to a window and saw the menu labels "include", "place" and "insert", they would likely search all of these menus to ensure that they had selected the correct item.

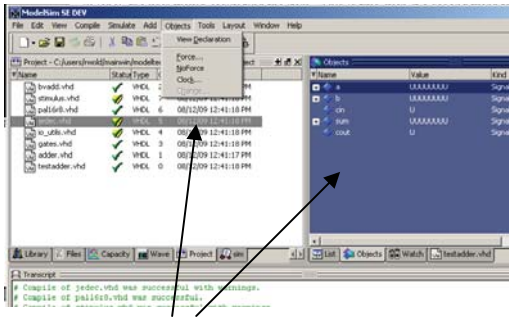
Partially Dynamic Menu Items – As Johnson [4] points out, menu items should not change and they should always be visible. Users expect the "save" menu item to be in the same location within the menu, regardless of what window is active. Through experimentation, we discovered that as long as a menu item's location remained constant, as well as the first word of the menu label, the user would easily find the item. This is true even if a second word to the item was added that more closely defined the operation. For example, Modelsim had windows that used the menu name "write", such as "write report" and "write file". Modelsim also had windows that used the term "save", such as "save format" and "save changes". We combined all of these menu items and, based on which window was active, changed only the second word of the item. Depending on which window is active, the save command label could be "Save Report", "Save File", "Save Format" or "Save Changes".

Active Window Menu – Through careful categorization and using partially dynamic menus we were successful in consolidated the majority of menu items. However, we found that most windows defined a set of commands that

were unique and could not be generalized or and combined with an existing menu pick. This issue was resolved by adding a main menu pull down that was specific to the active window.



When the project window is active, the active window menu lists project specific items.



When the objects window is active, the active window menu lists object specific items.

Figure 4

The active window menu text changes each time a new window is activated. The label text displays the name of the window and is useful for quickly determining which window is currently active.

3.0 Window Layout

For many years, the Modelsim GUI was based on a paned widget. As there were only a few windows, the ability to resize each pane was the only feature needed (figure 5).

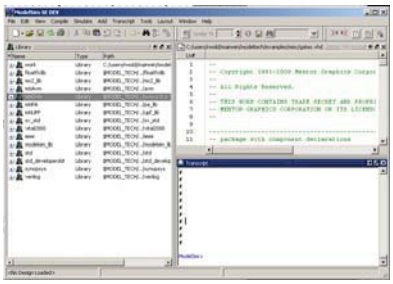


Figure 5

As Modelsim added support for more HDL languages and advanced debug features, the number of windows multiplied along with and the number of windows that a user typically opened. When a user opened a new window using the paned widget, screen real estate was taken away from the other windows to make room for the newly opened window. As the number of panes increase, the paned widget eventually became cluttered and some panes become too small to be useful (figure 6).

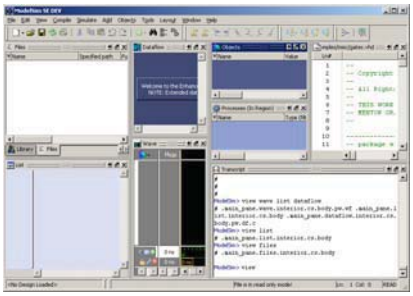


Figure 6

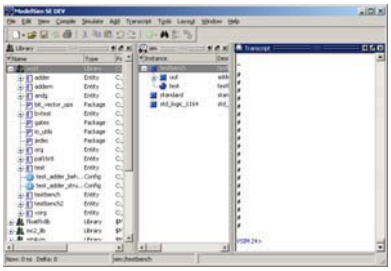
It was evident that the paned widget was an incomplete solution when many windows were opened. Customers complained that Modelsim lacked layout flexibility and that the tool was not using the limited screen space effectively.

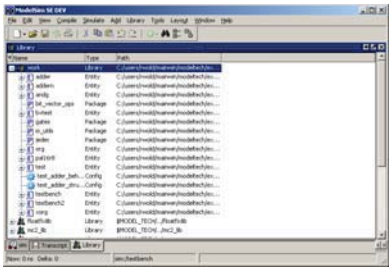
In order to resolve the issue, we first needed to understand what a user's ideal layout might be. Over a two year period we gathered information from customers on how they wanted their windows organized. At the Mentor Graphics User2User[3] conference we placed users in front of various window proto-types and asked them to open windows that they commonly used and organize them in fashion that worked best with their usage model. Based on this study, some interesting facts emerged.

-Users typically have a strong opinion on how windows should be placed and organized.
-User opinions vary dramatically on how windows should be placed and organized.
-A user's ideal layout will change depending on how they are using the tool.

The key discovery was that while window layout is important, there is no single layout that meets everyone's needs. Window organization and placement is a personal preference.

In addition, we found that customer layouts involved placing a window in one of three basic states, paned, tabbed or stand-alone.


Paned Windows


Tabbed Windows

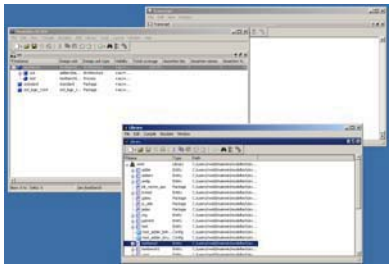
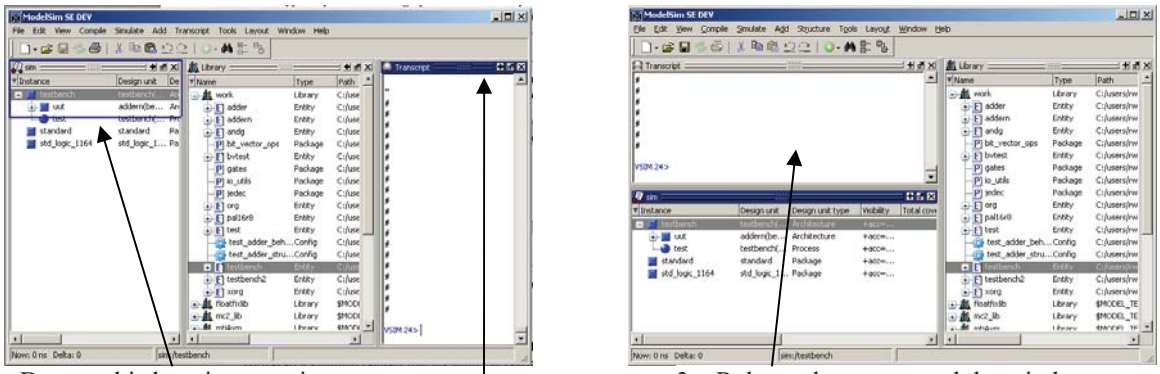

Stand Alone Windows

Figure 7

When asked to describe their ideal window layout, the majority of users defined layouts that used a combination of these states. Supporting all three of these states was a clear requirement as well as providing an easy to use interface for manipulating the layout.

3.1 Paned Windows

A common request with paned windows was the capability to rearrange the panes. We achieved this feature by extending the paned widget to support dragging and dropping of panes. Now, when a window's header (or more specifically a pane's header) is selected and dragged, drop zone highlighting appears under the mouse. This highlighting indicates that the window will be placed at this location should the mouse button be released.



2 – Drag to this location – notice the blue drop zone highlight

1 - Selecting the header

3 – Release the mouse and the window pane has been moved to the new location

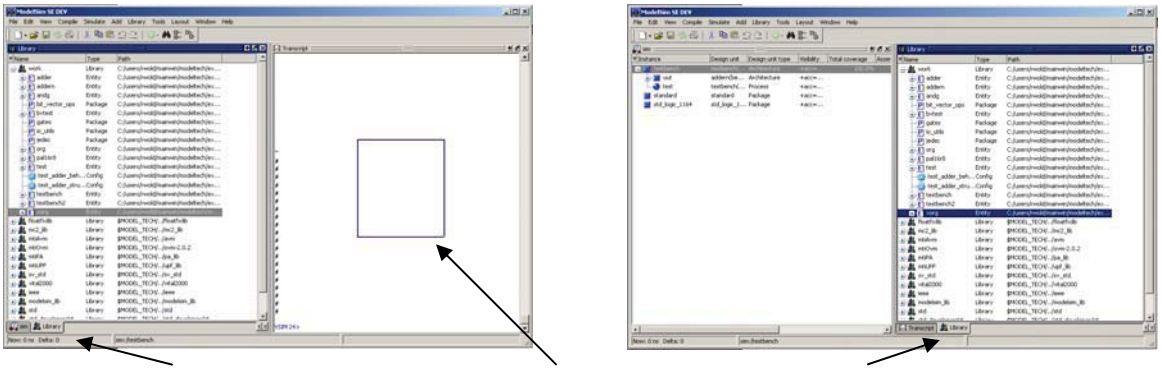
Figure 8

Each pane has a drop zone above and below as well as on the sides. There are also drop zones on the outer edge of the main pane. Given this combination of drop zones, users can produce any pane layout they wish.

3.2 Tabbed Windows

Tabbed windows are a popular way of displaying multiple windows while reducing the amount of screen real estate required. A tab group is a paned window with a group of tabs at the bottom. Selecting a tab raises that tab's window. Through discussions with users we discovered one obvious draw back with tab groups. Visibility of members of a tab group is mutually exclusive; a user can only see the contents of one window at a time. Plus, we found that there are no ideal tab groupings. Regardless of which windows are placed in the tab group, there will be a set of users that dislike the grouping and want to see the grouped windows simultaneously.

The resolution to this issue was simple: allow users to define their own tab groups. Using the existing pane drag functionality, we added a new drop zone to on the center of a paned window. When a window is dropped at this location, the destination pane adds the dragged window into its tab group. If the destination pane is not a tabbed group, it becomes one.



1- Select the windows tab and drag to a new location

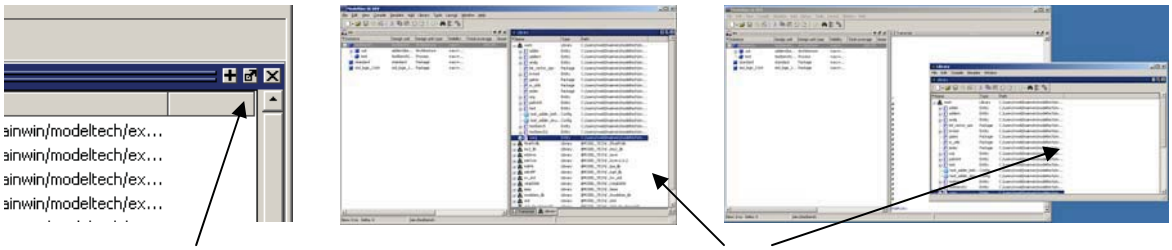
2- The drop zone highlight indicates the window will be placed here

3- The resulting drop forms a new tab group

Figure 9

3.3 Stand-alone Windows

Certain users are accustomed to working with windows that are separate and stand alone on the desktop. Modelsim supports this usage mode by allowing a window to be "undocked" from the main window and placed as a stand alone window.



1- Selecting the undock button on the window header undocks the window.

2- The window is removed from the main pane and becomes a stand-alone window on the desktop

Figure 10

3.4 Other Layout Features

The three window states described here provide the flexibility needed to achieve each user's custom layout. In addition to these features, we found that a user's ideal layout changes depending on how they are using the tool. If a user is running a simulation they are interested in one set of windows. If they are examining code coverage they are interested in a completely different set of windows. A window that has very important data in one phase of the project may become of little interest in a different phase of the project.

Users can save layouts and restore them, but we found it very useful to automatically select a users' layout based on how the tool is being used. If a user turns on code coverage, for example, Modelsim loads the layout that was last used when code coverage was on.

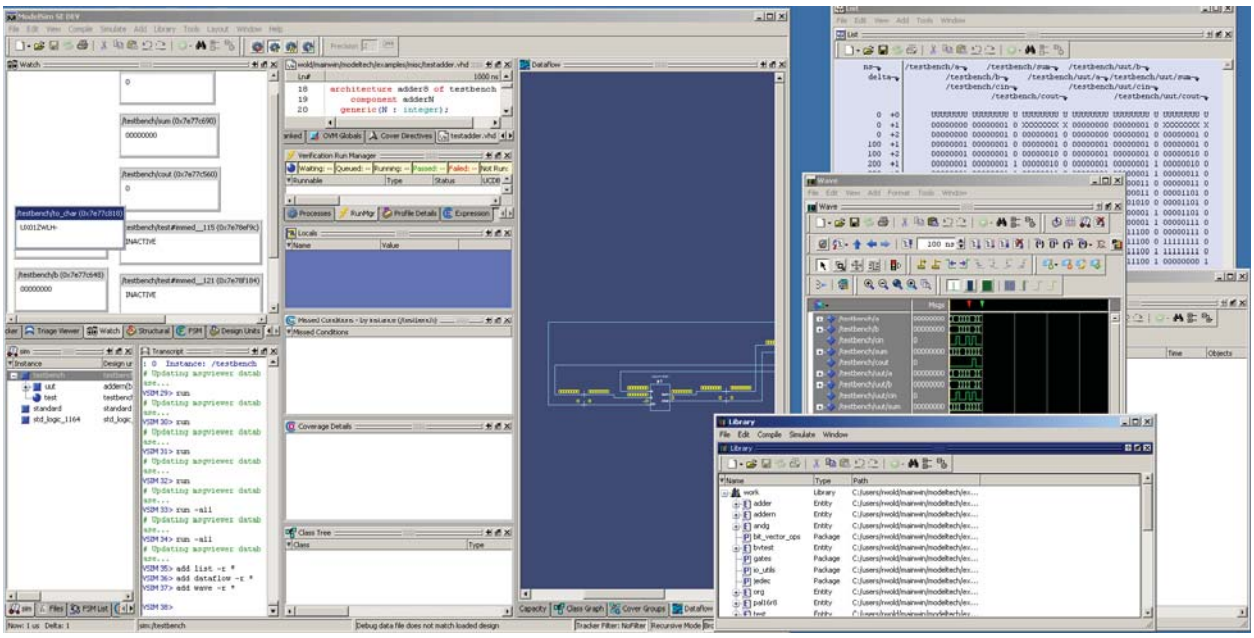


Figure 11 - The possibilities are endless....

4.0 Conclusion

As the number of windows within a tool increase, certain issues appear that are not experienced by a tool with few windows. These issues appear slowly and grow worse as the window count increases. The approaches described here have been implemented within Modelsim. They have reduced window development time and have been well received by users of Modelsim.

5.0 Acknowledgements

Special thanks to the Tcl/Tk community for providing a powerful, flexible, and portable GUI platform. Also special thanks to Brian Griffin who had a hand (or both) in identifying, developing and implementing the solutions described here.

6.0 References

[1] Doulos, A Brief History of VHDL, http://www.doulos.com/fi/desguidevhdl/vb2_history.htm.

[2] Doulos, A Brief History of Verilog, http://www.doulos.com/fi/desguidevlg/vb2_history.htm

[3] Mentor Graphics User2User conference. Mentor Graphics Corporation 8005 SW Boeckman Road Wilsonville, OR 97070, 2007 & 2008.

[4] Johnson, Jeff, GUI bloopers: Don'ts and Do's for Software Developers and Web Designers, Morgan Kaufman Publishers, San Francisco, (2000), pp 66.

[5] Miller, D. (1981). The depth/breadth trade-off in hierarchical computer menus. Proceedings of the Human Factors and Ergonomics Society 25th Annual Meeting (pp. 140-200) Santa Monica, CA.