Building Mission-Critical CAD Applications with Tcl/Tk

Michael McLennan (mmc@cadence.com)
Cadence Design Systems, Inc.

Abstract

The Signalscan waveform viewer has been an extremely successful product for many years, but it reached a point in its life cycle where it needed to be rewritten. In this paper, we'll describe how we used Tcl/Tk to rebuild and revitalize this mission-critical tool.

Introduction

These days, integrated circuits are designed and tested in software, long before any chips are actually produced. Designers use a Hardware Description Language (HDL) such as Verilog or VHDL to write code that describes the functionality of their chip. They use special-purpose computer-aided design (CAD) tools to compile their HDL code, simulate it, and convert it to the actual layout of transistors for their chip.

Complex designs have millions of transistors with millions of connections between them. The signals on the various connections must be monitored during the debugging process to ensure that the chip is working correctly. Each signal might fluctuate on the time scale of nanoseconds (10^{-9} s), and a simulation might run for many milliseconds (10^{-3} s). So it is not uncommon for a signal to have millions of high/low transitions, and for designers to look at several hundred signals in one sitting. Needless to say, any tool that plots that much data must be built in an extremely careful and efficient manner.

Signalscan is such a tool. Signalscan has been in widespread use in the integrated circuit design business for many years, and it brings in millions of dollars in product revenue each quarter. The code base had reached a point, however, where it had become increasingly difficult to maintain, and nearly impossible to enhance. Signalscan needed to be rewritten from the ground up, but the end result had to be robust, high-performance software that would continue to command millions of dollars in revenue.

To rebuild this mission-critical tool, we chose a development platform that is robust, object-oriented, cross-platform, and state-of-the-art in speed and functionality. In short, we chose Tcl/Tk.[1]

Some people think of Tcl/Tk as another VisualBasic—as a graphical user interface builder for toy projects. But it is much more than that. Tcl/Tk represents a paradigm for building software with a mixture of different components, often written in different languages, all glued together with a little Tcl code. In this paper, we'll show how we used Tcl/Tk on many levels—as a repository of contributed widgets, as a platform for building our own customized plotting widgets, as an object-oriented application language, and as a language for customer extensions.

1. The Tcl/Tk Paradigm

Tcl/Tk applications are a mixture of Tcl and C code. Learning how to apply the two different languages is difficult at first, but once mastered, it becomes a powerful new paradigm for building software. In a nutshell, C code is used to create solid building blocks with good performance, and Tcl code is used to coordinate the C code in a flexible manner. The result is a blend of speed and flexibility that cannot be achieved with any single language.

Let's see how this dual-language approach applies to the new Signalscan tool, which we currently refer to as SignalVision. Figure 1 shows the SignalVision waveform window. The plotting area on the right-hand side, which consumes most of the window, is composed of three custom-built widgets. The `markeraxis` widget (on top of the three) shows the time scale and a number of important time points marked with little flags called *markers*. The `signalbox` widget (in the middle) plots a list of logical signals and bus values. The `zoomaxis` widget (along the bottom) shows the current view relative to the entire time scale. It is a glorified scrollbar, with the time axis and markers drawn in the trough. It also has extra controls which allow the user to adjust not only the bubble, but the edges of the bubble as well.

All of these widgets are implemented in C code, so they are all highly optimized. But they are created and coordinated via a few simple Tcl commands. For example, a `signalbox` is instantiated with the following Tcl code:

```
signalbox .sbox -background {black gray}
grid .sbox -row 1 -column 2 -sticky nsew
```

The widgets communicate with one another by sharing data in a Model-View-Controller (MVC) pattern [2]. For example, the `signalbox` and the `markeraxis` widget sitting above it share a common set of markers. When a marker appears in the `markeraxis` widget, the line must also extend down through the `signalbox`.

Markers are stored in a `markerdata` object. Like the widgets, this object is implemented in C code, but controlled and connected to the widgets from the Tcl level. For

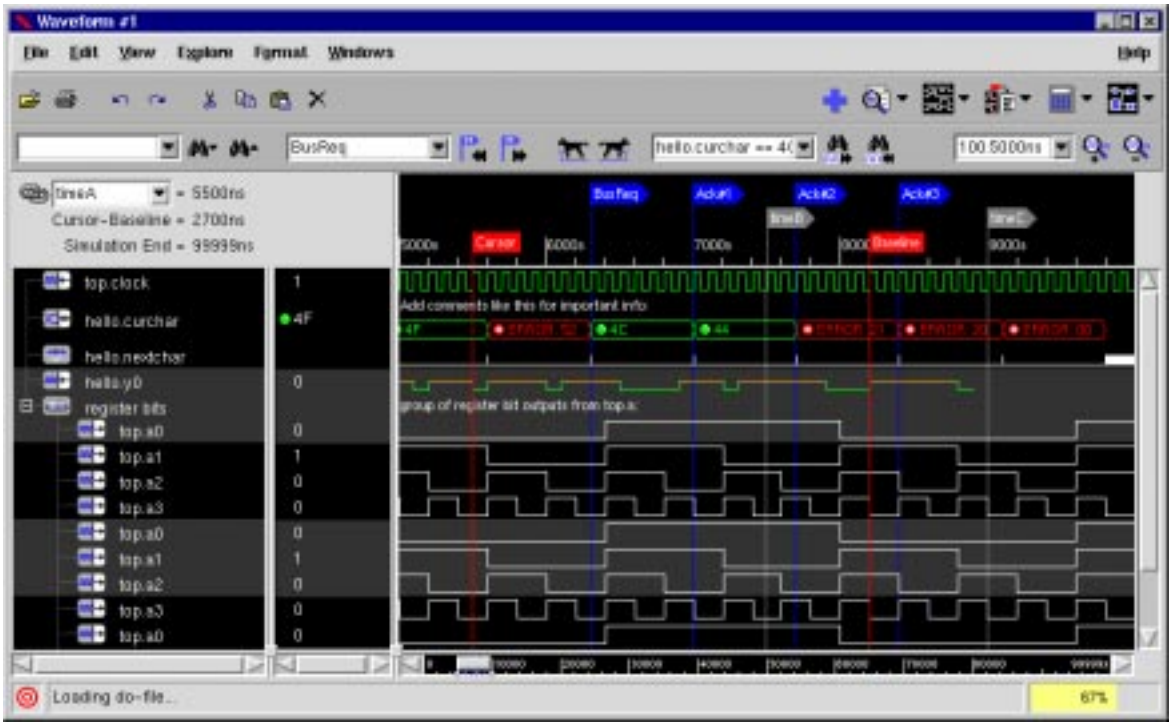

Figure 1: Main window of the SignalVision waveform plotting tool. The tool is a mixture of custom-built widgets, BLT widgets, and [incr Tcl/Tk] mega-widgets, all stitched together with Tcl code.

example, a `markerdata` object is created and plugged into the widgets as follows:

```
markerdata md
.sbox configure -markerdata md
.maxis configure -markerdata md
```

A marker is created with the following Tcl code:

```
set m [md marker create positional \
        -time 1000]
```

As soon as the marker is created in the `markerdata` object (model), the widgets (views) receive notifications and redraw themselves with the new cursor. The marker creation, notifications, and redraw operations execute at the speed of C code, even though we used a Tcl command to initiate the whole operation.

If any other part of the application is interested in marker notifications, it can register to receive them. For example, suppose we have a combobox full of marker names. It must be updated whenever a new marker is created. We could bind to the marker creation event as follows:

```
md notify add combo !markerCreate fixit
```

This creates a notification called `combo` on the marker creation event for the object `md`. Whenever the event occurs, the `fixit` command will be invoked to handle it.

Extra arguments are appended to the `fixit` command to communicate information about the event. Each event has different data associated with it. When a marker is created, for example, the callback receives the keyword `marker` followed by the marker name. This data is usually absorbed as `args` and converted to a Tcl array for processing:

```
proc fixit {args} {
  array set info $args
  puts "created marker $info(marker)"
}
```

Here is another example of a notification which is handy when debugging the application:

```
md notify add debug !all echo
```

This adds a notification tagged with the name `debug` to all events on this `markerdata` object. Any event will trigger a call to the `echo` procedure, which might be implemented as follows:

```
proc echo {args} {
  puts "EVENT: $args"
}
```

To turn off debugging, we simply remove the notification:

```
md notify remove debug
```

Getting back to our plotting widgets, we can add a little more Tcl code to control how the user interacts with the markers. This is the "controller" part of the MVC pattern. For example, the following commands allow the user to click the mouse to reposition the marker we created earlier:

```
bind .sbox <ButtonPress-1> {
  set time [.sbox convert x2t %x]
  md marker configure $m -time $time
}
```

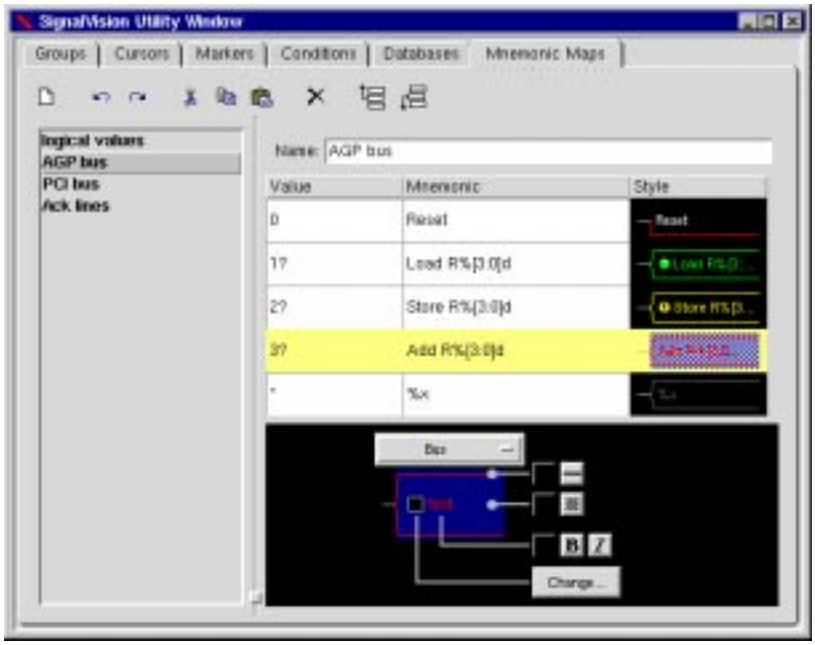

Figure 2: Utility window of the SignalVision waveform plotting tool. Pages are organized in a BLT tabset widget. The table shown here is a BLT hiertable widget with the hierarchy column hidden.

Having the "controller" part in Tcl makes it easy to change the behavior for different tools, or for different modes of the same tool. By separating the behavior of the widgets from their implementation, we have achieved considerable flexibility with no significant loss of performance.

2. Integrating Other Toolkits

Tcl/Tk has a large number of contributed widgets and extensions, and we used some of these to build our application. For example, the signals in the plotting area are stored in a `blt::tree` data object from the BLT toolkit [3]. This tree has a rich, C language API which supports our high-performance plotting, as well as a flexible Tcl interface. The tree is displayed on the left side of the window by a `blt::hierbox` widget, which we modified slightly to handle our own special requirements.

The Utility window shown in Figure 2 is also composed of BLT widgets. This window acts as an editor for the properties of all objects in the application. The various object types are organized into pages by a `blt::tabset` widget; selecting a tab brings up the editor page for that object type. The dotted line below the tab allows you to tear a page out of the notebook and onto the desktop. That way, you can cut and paste between different object types, or edit multiple object types at the same time.

The *Mnemonic Map* page in the Utility window contains a table of entries which define the map. If a signal value matches the glob-style pattern in the *Value* column, it is displayed by using the string in the *Mnemonic* column with the colors shown in the *Style* column. The style for the selected row can be edited by adjusting the controls in the preview area at the bottom of the window.

The mnemonic map table was implemented by using the `blt::hiertable` widget. This widget normally shows a hierarchy with plus/minus controls on the left, and a series of columns with related information on the right. We've configured the widget here to suppress the hierarchy column, leaving the basic table. We chose this widget instead of the TkTable package [4] for a number of reasons: It does a better job of resizing itself to properly display its contents; it has an underlying tree object to support its data, so rows can be inserted/deleted more easily; and it supports smooth scrolling.

We also borrowed a few widgets from the excellent BWidget package [5], we created custom mega-widgets with the [incr Tk] package (described later), and we added drag&drop support via the Tkdnd package [6].

Having a large library of OpenSource extensions was a tremendous boost to this project. The various packages were all quite useful, but we could not have built this application without the BLT toolkit. In our opinion, it is the best of the widget packages available for Tk.

3. Object-Oriented Development

Most of our plotting data is stored in custom-built data objects such as the `markerdata` object or the `blt::tree`. But plotting is just a small part of our overall application. There are toolbars, preferences, and scores of dialogs. If we were to build customized data objects for each part, we would quickly run out of time and funding.

```
itcl::class Toaster {
    inherit Notifier

    public variable heat 3

    constructor {args} {
        register !crumbs -fireonbind
        register !burn
        eval configure $args
    }

    method toast {nslices} {
        if {$crumbs > 50} {
            trigger !burn current $crumbs heat $heat
            error "== fire! fire! =="
        }
        set crumbs [expr $crumbs+$heat*$nslices]
        trigger !crumbs current $crumbs
    }
    method clean {} {
        set crumbs 0
        trigger !crumbs current 0
    }
    private variable crumbs 0
}
```

Figure 3: Simple example of an [incr Tcl] class which supports notifications by using our Notifier base class.

Instead, we used the [incr Tcl] extension [7-9] to handle most of our data at the Tcl level. [incr Tcl] extends the Tcl language to support object-oriented programming. It has single and multiple inheritance, virtual methods, static data members and member functions, and so forth. It is patterned after C++, but integrated into the Tcl mindset, so many people find it easy to learn and a natural fit with the rest of Tcl/Tk.

[incr Tcl] also supports our performance requirements. All member functions are byte-code compiled, and access to data members is highly optimized. So the overall performance of [incr Tcl] objects was more than sufficient for much of our application. [incr Tcl] also allows you to integrate C code with your Tcl code, so while a class might be defined at the Tcl level, its methods could be implemented in C. This not only helped us optimize critical parts, but also helped us raise useful C functions to the Tcl level.

For example, we have a small library of C functions that support the `notify` methods described in Section 1. We wanted to use the same `notify` methods for all objects in our application—including those implemented at the [incr Tcl] level. So we created a `Notifier` base class. This class is defined at the [incr Tcl] level, but its methods are implemented in C, and therefore have access to our notifier C library.

Having defined this base class, we can create many derived classes that support notifications. Consider the simple `Toaster` class shown in Figure 3. This class has four events: `!configure`, `!destroy`, `!crumbs` and `!burn`.

The `!configure` event is inherited from the `Notifier` base class. It is triggered automatically whenever a public variable is modified via the `configure` method. The `!destory` event is also inherited. It is triggered automatically when an object is being destructed.

The `!crumbs` and `!burn` events were added specifically for Toasters. These new events are registered in the Toaster constructor. The `!crumbs` event is registered with `-fireonbind`, so any client that adds a notification for `!crumbs` will immediately be notified of the current crumb count. This is similar to the way a Tk listbox notifies a scrollbar of its current view whenever the `-yscrollcommand` option is set.

Once registered, the events can be triggered by any method within the class. The `toast` method, for example, always triggers a `!crumbs` event to signal a new crumb count. Two bits of information are added to the event: the keyword `current` and the current crumb count. Suppose you were to create a Toaster and bind to its `!crumbs` event as follows:

```
proc show_crumbs {args} {
    array set info $args
    puts "crumb count: $info(current)"
}
```

```
Toaster fred -heat 4
fred notify add main !crumbs show_crumbs
```

Since the `!crumbs` event was registered with `-fireonbind`, this last statement would trigger an immediate call to `show_crumbs` as follows:

```
show_crumbs current 0
```

This would send the following message to standard output:

```
crumb count: 0
```

Further calls to the `toast` method would cause additional notifications:

```
% fred toast 2
crumb count: 8
% fred toast 1
crumb count: 12
...
```

At some point, the Toaster's crumb tray would fill to its limit, and the `toast` method would trigger a `!burn` event, including the current crumb count and heat setting as additional parameters.

We used the same principles to create many other kinds of data objects for the SignalVision application. For example, the list of mnemonic maps (shown on the left-hand side of Figure 2) is stored in a `ListManager` class. When a new map is added to the list, the `ListManager` triggers a `!insert` event. Many different parts of the application receive this event—including the listbox in the *Mnemonic Maps* tab, the *Format* menu on the main window, and so forth—and they update themselves accordingly. Having data objects which support these kind of notifications proved to be invaluable. It provided a simple way to manage the complex interactions in this gigantic tool.

4. Mega-Widget Development

[incr Tcl] also supports mega-widget development [10]. Mega-widgets are new classes of widgets constructed with Tk widgets as component parts. They look and act just like Tk widgets, but they are created with [incr Tcl] code instead of C code. As a result, they can be created in a fraction of the time it took to create custom-built widgets, such as the `markeraxis` and the `signalbox`.

The combination of a `markeraxis`, a `signalbox`, and a `zoomaxis` widget is an example of a mega-widget. Having this mega-widget makes it easy to create this stack of widgets and set up their default bindings. This, in turn, makes it easy to create two such stacks and put them side-by-side, thereby implementing a split-screen capability.

We've implemented many other mega-widgets as well, including comboboxes with type-ahead search capability, buttons with a drop-down menu of variant choices (like the history buttons in Internet Explorer), listboxes with automatic scrollbars that pop up as needed, progress bars, paned windows, and so forth.

5. Multi-Level Undo

Multi-level "undo" was another requirement for our new tool. We implemented "undo" by leveraging the power of Tcl. We keep two variables at the Tcl level: an "undo" stack and a "redo" stack. Each stack is a list of Tcl commands needed to undo/redo an operation. For example, suppose the user deletes a signal. The commands to delete and recreate the signal are added to the "undo" stack as follows:

```
set redoCmd {signal_delete top.a}
set undoCmd {signal_insert top.a 10}
lappend undoStack [list $redoCmd $undoCmd]
```

When the user invokes *Undo*, we grab the last pair of commands from the `undoStack` variable and invoke the `undoCmd` part. Then, we push the set of both commands onto a `redoStack` variable. When the user invokes *Redo*, we grab the last pair of commands from `redoStack`, invoke the `redoCmd` part, and push the commands back onto `undoStack`.

Since the whole application is built with Tcl commands, there are no limitations on what can be undone or redone. We have a rich syntax for logging and replaying operations that we didn't have to invent. It comes for free with Tcl.

6. End-User Customization

Many users like to customize a tool by adding their own buttons and panels, thereby creating their own "macros" for common operations. In some tools, this capability is limited to a few buttons or a customized menu, which can be programmed from a limited palette of operations.

Since our tool is built with Tcl, we already have a rich syntax for controlling the tool. We also have an extensive, well-documented collection of Tk widgets that we can expose to customers. We must be careful, however, to make sure that our graphical interface is insulated from any code that the customer supplies. If the customer were to set a variable in their code, for example, it can't crash our GUI.

We can protect our code by executing any customer code in a separate, "safe" interpreter [11]. Tcl supports multiple interpreters in any one application. Each interpreter has its own set of commands, variables, and widgets. We can insulate the customer from details of our implementation by providing a well-documented Application Programming Interface (API) of commands in the "safe" interpreter. This API includes commands to add buttons to the main window, and to create separate pop-up windows of user-defined controls. So customers can add new features, and these features are integrated seamlessly into the tool.

Customer-supplied modules are treated as "plug-ins" for the application. Plug-ins can be enabled and disabled from the Prefereces dialog.

Conclusion

Layering Tcl/Tk code on top of C creates a potent mixture of speed and flexibility. It has allowed us to offer advanced features such as split-screen capability, multi-level undo, and end-user customization with little extra effort. The expressiveness and flexibility of Tcl has allowed us to rewrite several hundred thousand lines of code—more than 20 man-years of effort—in a little under a year. The finished product offers the same level of performance as the original tool, but much more flexibility. Tcl has given our product new life, along with a solid code base that can be extended for many years to come.

References

[1] John K. Ousterhout, "Tcl and the Tk Toolkit," Addison-Wesley, 1994. Tcl is available as OpenSource from http://sourceforge.net or http://dev.scriptics.com.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, Reading, MA, 1995.

[3] The BLT Toolkit was created by George Howlett, and can be downloaded from http://www.tcltk.com/blt/

[4] The TkTable package is available from http://tcl.ActiveState.com/community/hobbs/tcl/capp/

[5] The BWidget package is available from http://sourceforge.net/projects/tcllib

[6] The Tkdnd package is available from http://www.iit.demokritos.gr/~petasis/tcl/tkDND/tkDND.html

[7] M. J. McLennan, "[incr Tcl]: Object-Oriented Programming with Tcl," *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[8] Chad Smith, *[incr Tcl/Tk] from the Ground Up*, McGraw-Hill, 1999.

[9] [incr Tcl] is available as OpenSource from http://www.tcltk.com/itcl/ or http://sourceforge.net or http://dev.scriptics.com.

[10] M. J. McLennan, "[incr Tk]: Building Extensible Widgets with [incr Tcl]," *Proceedings of the Tcl/Tk 1994 Workshop*, New Orleans, LA, June 23-25, 1994.

[11] Mark Harrison and Michael McLennan, "Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk," Addison-Wesley, 1997.

Acknowledgments

Thanks to Ahran Dunsmoor, Tom Fitzpatrick, Mike Floyd, Doug Koslow, Mark Harris, Deb Mandel, Rick Meitzler, Mary Nguyen, and other members of our team for their contributions. Thanks also to Barb Henry, Sathyam Pattanam, and Rahul Razdan for supporting this work.