Tcl/Tk-based Alarm Presentation System

K.R. Fitch
Westinghouse Electric Company
1740 Golden Mile Hwy, Monroeville PA, 15146, USA
kfitch@notes.westinghouse.com

Abstract

This paper describes my experiences in implementing an alarm presentation system (APS) for a nuclear power plant using Tcl/Tk as the basis for the user interface. After defining just what an alarm presentation system is, I describe the basic architecture, and highlight how I use various features of Tcl/Tk in the design. The ultimate goal is to give the reader the sense of the power of Tcl/Tk for implementing complex applications and to help the reader learn from my experiences and mistakes.

1 WHAT IS AN ALARM PRESENTATION SYSTEM?

Nuclear power plants world-wide are upgrading their instrumentation and control (I&C) systems from analog to digital systems as part of plant life extension programs. Westinghouse Electric Company is currently implementing a major I&C and control room replacement project for the Ringhals 2 plant in Sweden. One important element of existing control rooms is the annunciator system, which typically consists of many hundreds of hardwired, backlighted, engraved alarm tiles arrayed in some number of alarm boxes. These tiles flash to alert the operator when alarms change state and are steadily lit when an active alarm is acknowledged.

An important human factors attribute of fixed tile layout annunciator systems is that the operators quickly learn to associate patterns of lit tiles with various common plant conditions. They can often recognize at a glance what is going on. The Ringhals 2 customer wanted Westinghouse to develop a software-based alarm presentation system (APS) which maintained the strengths of the fixed layout system while supporting "click on the tile" access to live data, alarm response procedures, and other reference data. The APS feature set was specified in considerable detail in the contract documents.

The control system hardware and software being used for the project is from the Ovation™ process control product line, which is supplied by Westinghouse Process Control (an Emerson company). The Ovation system is a distributed system which consists of controllers (custom PCs) that interface with plant I/O signals and of workstations (Sun Solaris and/or Windows NT systems) that provide the operators with control and monitoring displays. All of the controllers and workstations are connected by a high-speed data highway which lets them share data. An integrated alarm system is among the standard features of Ovation, but this "base alarm system" can only present the alarm data in a series of chronologically ordered lists. The software also provides a graphics language for generating graphic displays and animating them with live data. However it is limited in the number of graphic objects available on a single display and is not capable of supporting many of the desired APS features. The choice was made to investigate how far we could go in implementing APS using Tcl/Tk as the basis for the presentation layer. I became involved at this point because of my previous experience in developing several modest Tcl/Tk applications.

APS is what the nuclear power world calls category C software. (The categories are formally defined in reference [1].) Basically, this means that it is not safety-critical software, and that even if it breaks or fails there will be no threat to the public from radiation release. It may seem a little strange to define as non-safety critical an alarm system that tells the operator when something abnormal is happening. But as a result of the Three Mile Island accident in 1979, operators are now trained to deal with any severe problem by following emergency response procedures that focus only on the values and trends of a fairly small set of critical indications. Whether or not the alarms work really doesn't matter in a safety context. Because APS is category C software, there is no problem in using commercially available software such as Solaris and open source software such as Tcl/Tk.

Half of the APS described in this paper is scheduled to be installed during a control room upgrade in mid 2004, with the other half in the following year.

2 THE APS ARCHITECTURE

2.1 The Customer's Vision

Long before the Ringhals 2 upgrade contract was awarded to Westinghouse, the control room operators had been thinking about how they wanted to interact with a new software-based alarm presentation system. They knew what they had in the existing plant, and they knew the kinds of information they needed to have to respond to alarms. They also knew the limitations of the previous generation of the Ovation list-based base alarm system, and had researched the approaches that others were using in their plants.

Because they had done their homework, the upgrade contract specified what they wanted in considerable detail, right down to the wording of the menus and the layout of an idealized large screen display. There was a large gap between the system we had in the standard product and the system they wanted, and the challenge for Westinghouse was to see what we could do to close the gap.

In hardware space, the key element of the specification was the desire for a single large video display unit (VDU) to be controlled by a single mouse/keyboard. This VDU had to fit in a control room console and not obscure the view of the "back panels". It had to be seismically qualified to the extent that it wouldn't fly to pieces during a design basis earthquake. It had to have a sufficiently large display area capable of depicting all

of the key alarms simultaneously in a tile-based format with a character height of no less than 4 mm (This is the human factors-based minimum size for readability from one meter away). To maintain similarity with the alarm tiles in the existing plant, each tile was to be capable of displaying four lines of twenty characters each. There are on the order of several hundred key alarms associated with each of the four major plant subsystems (the reactor, the twin turbine-generators, and the electrical system). In practice there is no single display unit available that can handle the necessary number of pixels.

In software space, the key elements were as follows. First, the system had to allow for a hierarchy of alarms, where less critical alarms could be logically combined to form a single high level alarm that depicts the most severe underlying alarm state. Information overload is a significant human factors problem in plant upset conditions that can cause hundreds of alarms to occur in quick succession, and this hierarchy is a technique for dealing with this problem. When the high-level alarm comes in as an isolated event, the operator can immediately click on the alarm tile to bring up a display of the alarms that comprise the high level alarm. In the immediate aftermath of a major plant upset where many of the inputs to the high level alarm may go into alarm, the less critical alarms won't place undue demands on the operator's attention.

Second, the operator needed to be able to have immediate online access to the procedures and technical information he needs to deal with each alarm. The existing binders of paper procedures were to be replaced with HTML procedures, complete with links to auxiliary information (e.g. technical manuals for pumps). The kinds of technical information needed include summaries of the alarm limit set points, trend plots of the process signals that are the inputs to the alarm, and access to the database containing all of the information about the point. In addition, they wanted the software equivalent of a Post-It™ note so that they could leave information about the alarm for the next shift of operators or the maintenance staff.

2.2 Hardware Architecture

The overall I&C system being developed by Westinghouse for the Ringhals 2 plant uses the Ovation family of hardware and software. In this implementation, the Ovation system uses Sun workstations running Solaris to provide the operator interface. There are four instances of APS in the Ringhals 2 control room, one for the reactor area, one for the electrical area, and one each for the twin turbine-generators. APS runs as an application on four of these Solaris workstations, with each instance showing a different set of alarms. Because a single VDU is not sufficient to present all the information at once, we use a system called EOS (made by BARCO - see reference [2]) to create a single logical large screen X Windows server out of six physical 1280x1024 VDUs. These VDUs are arranged in a three columns, two rows shape (see Figure 1 below). The APS application then remotely displays its Tk toplevels on its associated EOS X Server. The EOS X Server allows the mouse cursor to move freely over the entire logical screen. A single keyboard is also available, but it is used only on a limited basis for text entry. Figure 10 (at the end of this paper) shows the APS as it would look on a seamless 3840x2048 display.

Figure 1 – A Close-Up View of APS

Figure 1 is an example of one instance of APS as shown in a control room mockup. There will be four such APS stations in the control room. Note that the angled arrangement is driven by the need for the operator to be able to see over the top to the "standup" panels.

Figure 2 - A Wide Angle View of the Control Room

In the mockup photo in figure 2, the reactor area APS instance is in the right center of the picture. One or more operators will use the APS while seated or standing. Different alarm sounds for each plant area and a flashing light at each APS station are used to quickly orient the operators (who may be away from the console) when an alarm occurs.

2.3 Software Architecture

In terms of software, APS makes use the base Tcl/Tk code (version 8.3.3) as well as tclX (8.3), tclhttpd (3.2.1), itcl (3.2), and tcllib (0.8) all running on Solaris 2.6 and compiled using the Sun C compiler. The only non-standard configuration parameter in the configure/make process is the use of the SECURITY_FLAGS=-DTK_NO_SECURITY option. We do this to avoid the constraint that "xauth" security be used to enable the Tk send command. The xauth security model is not consistent with the xhost-based scheme that is pervasive in the Ovation system. Note that the plant network is carefully isolated from uncontrolled systems on the plant network or the Internet.

The APS application is single-threaded and uses the Tcl event loop as the master scheduling method. It uses a variety of customized Tcl extensions to interface with the Ovation system. Included with the Ovation system are a number of application programming interfaces (APIs) that allow programmatic access to plant data as well as control of various Ovation displays and features (e.g. trend displays, database lookups). Tcl interfaces to all of the potentially interesting ones were developed, whether or not they were initially used by APS or not.

APS runs at a nominal frequency of 2 Hz. This rate is driven primarily by the need to flash alarm tiles with a one-half second on/one-half second off pattern. The input signals to the alarm tiles are typically scanned by the Ovation system at a 1 Hz rate. Because the amount of time needed to process the alarms will vary from cycle to cycle, we read the system clock when we start processing, read it again when we're done, and then wait using the after command for a period of time based on the difference. A transient condition where the compute time is more than the allotted one half second is not reported as an error, while sustained use of more than the allotted time is.

The fundamental unit of the APS is the alarm tile. All of the configuration data for each tile is accessed by using the alarm tile ID as the key to a set of Tcl arrays. Thus there is a row array, a col array, a tile_text array, and so on. Each non-blank alarm tile has a "alarmtile" widget (an adaptation of the standard button widget) and (depending upon the type of tile) a menu widget associated with it.

Alarm boxes and alarm lists are composed out of a rectangular array of alarm tiles. Each alarm box/list is associated with a separate Tk toplevel to allow the boxes/lists to be positioned on the display(s), iconified, and withdrawn independently. The Tcl grid geometry manager is used to arrange the tiles within the boxes, while the ability to embed widgets in a text widget is used for lists.

APS is a hybrid of Tcl code and C language extensions. Since the Ovation API is written in C, we needed extensions to read values and alarm state information. Beyond that, the question arose of how much of the work to do on the Tcl side, and how much to do on the C side. The partitioning of the work was decided upon iteratively by trying a partitioning, measuring where the bottlenecks were using the TclX profiling package, and then replacing a frequently-used slow Tcl algorithm with a faster C one. The major performance improvement steps were to implement the alarm logic for each tile in C, to implement the loop over the entire set of tiles in C, and to pre-sort lists wherever possible.

The primary remaining performance bottleneck in the APS is the implementation used for flashing (blinking) of the alarm tiles. The tiles are blinked by synthesizing one or more "widget configure –bg blink_color" commands in a string and then using TclEval() to process the command string. In other words the Tcl code calls a C extension that internally runs Tcl code when necessary. The bottleneck can be significant if there are a large number of tiles blinking at the same time, and the CPU usage can double or triple. Some thought was given to reaching down into the internals of the widgets to directly manipulate their data structures to change the colors. Ultimately this was rejected both because it is a Bad Idea to stray beyond the bounds of the published API for Tcl/Tk (or anything else), and because it looked to be a significant amount of work. In any case, given the fact that the project is being developed over the course of several years, it was felt that it was a good bet that faster CPUs would come to the rescue (if indeed we needed to be rescued) once the final APS configurations were defined. As of now, APS is capable of supporting at least two thousand alarm tiles per instance, with one half of them blinking, while using about one-half of the processing power of an Ultra-5 workstation with an UltraSparc IIi 333 MHz CPU and 256MB of memory.

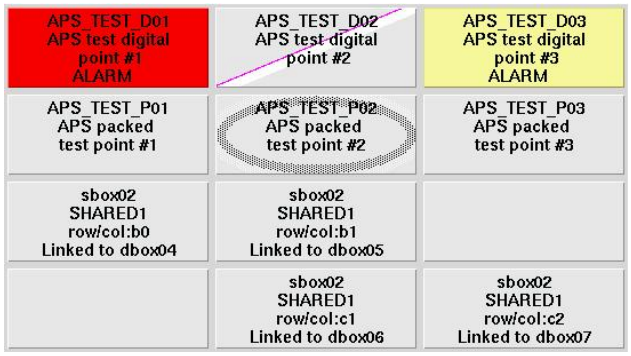

Figure 3 - A Typical APS Alarm Box

The typical alarm box shown in Figure 3 above gives you the idea about what the operator sees on the VDUs. Each tile in the grid either represents a single plant process point that can go into an alarm state or an aggregate point composed of an arbitrary set of plant process points. In the former case, the user can left-double-click on the tile to acknowledge the alarm and right-click on the tile to bring up a menu of supporting information. If the tile represents an aggregate point, a right-click brings up another alarm box or an alarm list that shows all the inputs. Holding down the middle mouse button brings up an enlarged view of the alarm tile.

The background color of the tile denotes the alarm state. A red background is a high priority critical alarm, a yellow background is a normal alarm, a green background is an alarm that has just cleared, and a gray background is an alarm that is off. A special magenta background is used to denote points that are out of service for one reason or another. Flashing tiles cycle between the gray off color and the alarm state color. We allow for a different foreground text color to be associated with each background color, and have made the color controls accessible via a Tcl interface. Currently we use black foreground text throughout APS. We've experimented with changing the text color to white for the red background case to improve the contrast, but the flashing between white-on-red and black-on-gray is disconcerting for some users (including the author).

Points that are connected to analog signals (e.g. a pressure transducer) can have up to ten different alarm conditions (four high levels, four low levels, and two user-defined levels). If desired, the particular level that is in alarm can be shown as dynamic text in the alarm window.

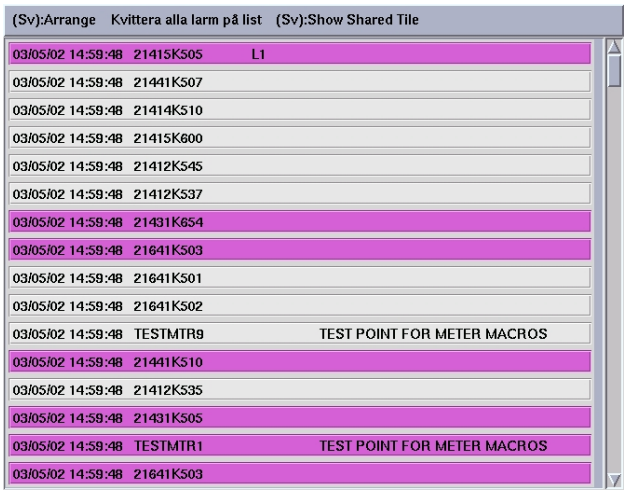

Figure 4 - A Typical APS Alarm List

The alarm list format is typically used for alarm aggregations that have too many inputs to be arrayed in an alarm box. In effect, it is simply a one column alarm box with differently shaped tiles. The scrollbar allows for navigating among a long list. Alarm lists can be rearranged by the operator, and the technique for doing so was a good example of the "brute force" approach. The alarm list is created as a set of alarmtile widgets (a modified Tk button widget – see below) embedded in a text widget. To rearrange these in chronological, priority, and alphabetical orders, the algorithm works through an ordered list of widgets. For each widget we save the widget coloration, destroy the widget, and then recreate the widget, inserting it at the end of the text. This accomplishes the sort.

Figure 5 - The APS master control display

The master control display performs several roles. It allows the user to select from a set of lists that show all the active alarms in various categories. It allows the expert user who knows a password to control the behavior of the APS. It has a master acknowledge button that lets the user acknowledge every alarm that is currently visible. And finally, it provides visual feedback that the APS is running and indicates how much CPU power it is consuming. Most of the time in the power plant there are no alarms changing state, and so the APS looks static. How then is the operator to determine at a glance whether or not the APS is still running? To provide this feedback, we move the bar inside the middle bar slowly from left to right. The color of the bar is determined by the worst alarm state in the entire APS. The width of the bar is made proportional to a smoothed value of the CPU power being used by APS. In figure 5 above, APS is using about 10% of the CPU power of the machine.

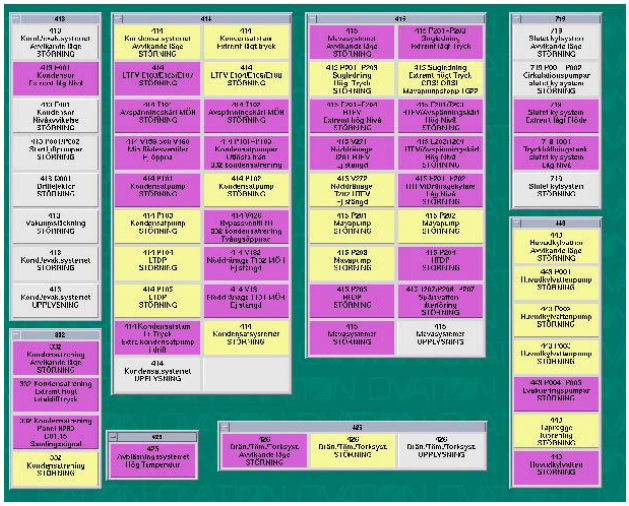

Figure 6 - A Set of APS Alarm Boxes

Figure 6 shows an arrangement of alarm boxes. Typically, each box groups alarms from one plant system. Such arrangements are permanently shown on four of the six VDUs, with the other two reserved for transient information, menus, trend displays, and the like.

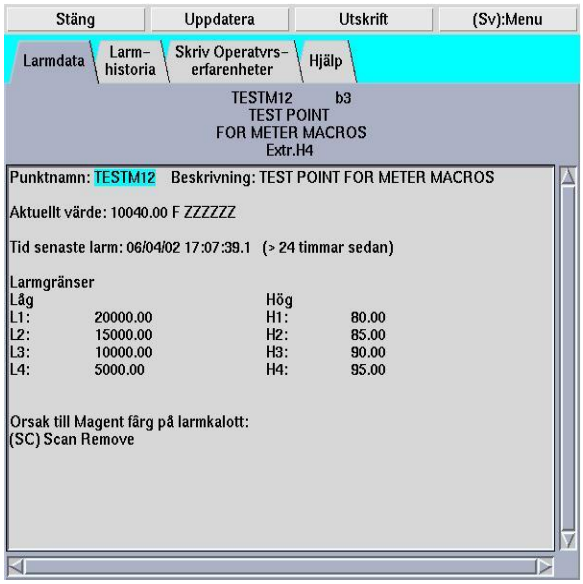

Figure 7 - The APS Background Information display

The APS background information display shown in Figure 7 uses the itcl tabnotebook widget as its basis. The operator brings up this display from the menu associated with each alarm tile. The tabbed notebook proved to be a very nice approach for making a lot of information available to the operator with a minimum of mouse clicks.

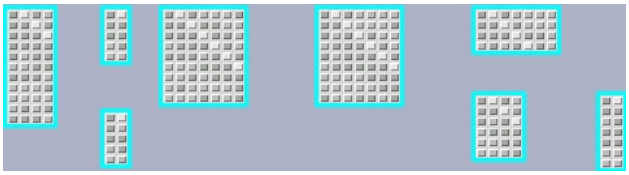

Figure 8 - The "Shadow" display

The "shadow" display shown in Figure 8 was developed as an experiment in allowing all of the fixed APS displays (four of the six VDUs) from all four of the APS instances to be shadowed (copied) on the control room shift supervisor's console. Each alarm tile is shown as a one character cell, with a non-blinking coloration corresponding to the alarm state. Clicking on the miniature tile provides all of the same functions as the full scale one, without consuming a large amount of screen space. The experiment worked in the technical sense, although I found it difficult to control the scaling and sizing such that the miniature boxes were positioned at positions corresponding to their full scale equivalents. In the end, the customer decided that they did not want to use the feature.

3 TCL/TK AS USED IN APS

3.1 The Catalytic Action of Tcl/Tk

Those readers who took a chemistry course somewhere along the way may recall the concept of "activation energy". Basically, in order for two chemical reactants to form a product, you have to add enough energy to push the reactants together (forming an "activated complex") long enough for them to react. If the activation energy is high, the reaction either takes place slowly or not at all. One way to lower the activation energy for a reaction is to add a catalyst. The presence of a catalyst makes it more likely that the reaction will proceed.

Tcl/Tk is a catalyst for encouraging experimentation with a GUI system such as APS. It is very easy to alter the visual elements (widgets, fonts, colors, menus, etc.) either by a quick edit of the *.tcl source files or even by interactively changing parameters while the program is running. The activation energy for just trying something out is much lower than it is in a more traditional edit/compile/make/execute environment.

Tcl/Tk (as well as the associated extensions such as incr Tcl) is also a catalyst for developing a complex GUI in the first place. It is a highly expressive language that allows you to tell the computer to do quite a lot of work with very little code.

The initial prototype of APS was ready to show to the customer in the space of two months, much of which was spent iteratively working on eliminating performance bottlenecks by replacing Tcl code with C code. At that point there was little in the way of serious error checking, and features like internationalization were missing. It quickly became clear that Tcl/Tk was capable of handling the design load and desired features of APS.

3.2 Tcl/Tk In Large Applications

Many different complex applications have been developed using Tcl/Tk, and no doubt there are larger, more complex applications than APS. Regardless, I have been quite impressed with how well Tcl can handle large applications with thousands of widgets and a single 3840x2048 logical screen comprised of six regular VDUs. I had some concerns about how well it would scale up, but found that Tcl/Tk just works for large applications. No fuss, no muss, no memory leaks over weeks of runtime, no problems handing a giant screen. There are of course minor quirks such as tk_dialog pop-ups that get centered in mid logical screen (and therefore straddle two physical VDUs), tearoff menus that get orphaned, and the occasional out and out bug (see SourceForge bug ID 483832) but these were easy to address or work around.

As it stands today, the APS is comprised of 17K lines of Tcl code (including blank lines and comments). There are 154 Tcl procedures defined. The configuration data that defines the layout and inputs to the alarm tiles when written as Tcl code (see below) varies based on how many tiles are configured, but is typically about 25K lines of Tcl code (with very few blank lines or comments, since this code is dynamically generated at APS startup and is not meant for human eyes).

When the system is running in a typical configuration, it uses 390 MB of virtual address space. One side effect of having such a large address space on a Solaris system is the requirement to have a larger than normal swap area that can accommodate two 390 Mb processes on a transient basis whenever the exec command is invoked.

3.3 Writing & Interpreting the program at runtime with the source command

One of the key self-imposed requirements on the APS was the necessity to defer as much as possible of the configuration processing until the APS instance is started. In the Ovation environment, software is maintained and distributed from a software server. For configuration control reasons among others, this software server typically does not even have a compiler. All standard Ovation applications are data driven in that they read configuration files as they start up. In the case of APS, there are four different instances with different sets of inputs and alarm tile arrangements, and it was simply unacceptable to have to maintain four separate versions with compiled-in configurations.

The "master" configuration information for APS resides in an Oracle database which encompasses essentially all of the data that configures the I&C system for the plant. Because of the architecture of the Ovation system, the database information is exported in the form of ASCII configuration files in formats dictated by the application program. For APS, I developed a simple file format designed to be easily parsed. This file contained in a compact form all of the information needed to arrange and group the alarm tiles on the "logical" display screens, the mapping information to associate physical X screens with the logical ones, and the unique identifier of the individual points that were "wired" to the tiles.

The general startup sequence for the APS is:

- Invoke wish with the "main" Tcl script file

- Source in the basic Tcl files for error logging features, etc.

- Source in the Tcl configuration parameter file that defines the generic behavior of APS (operating modes, directories where files & extensions reside, etc.)

- Load in the custom Tcl extensions

- Invoke Perl to read the alarm tile configuration parameter data and write a set of Tcl statements that correspond to the selected configuration. Typically a single input line of alarm configuration data causes eight or so lines of Tcl code to be written, and a full specification of an APS instance will have 3-4K lines of input.

- Source in the rest of the Tcl source code, establishing interfaces as needed to Netscape, Acrobat reader, and various Ovation applications.

- Source in a "last chance" Tcl file that potentially allows the user to override previously-defined values or procedures

- Use the Ovation pointname as a key to extract other point-related information from the Ovation database

- Pass the necessary data to the alarm system "compiler", which populates C language data structures that are used in the Tcl extension to greatly enhance the performance of the system.

- Create the widgets, menus, pre-sorted lists, etc., needed to run the system

- Start checking for alarms…

The entire startup process takes about thirty seconds. Why add an external invocation of Perl instead of staying with Tcl? Simply because Perl is better at doing "text transformations" than anything else I've used.

3.4 Developing Tcl Extensions

When I last used Tcl for an application in the early 1990s, I had built up a fairly trivial extension to interface with a simulator system, so I was familiar with the concepts. For APS, I needed two extensions, one to a large subset of the many routines in the Ovation API, and one to my own customized alarm processing code, which was used to dramatically improve the performance of the system. I looked at using the SWIG tool, but rejected it in part because the available include files and function prototypes for the Ovation API were something of a mess, and in part because I was going to have to customize the documentation format and error handling approaches anyway. It was simply easier to develop a template, and copy/customize the template as needed.

I believe that Tcl is one of the easiest languages to write and manage extensions for. It is certainly easier than writing extensions for Perl (at least for a beginning extension writer). The "everything's a string" concept as input, and the ability to pass back as output single long space-delimited strings and process them on the Tcl side either as lists or as array key-value pairs is very powerful. The cumulative upwards error reporting and the built in features of the function Tcl_GetIndexFromObj() are very useful and make the programmer's life simple.

3.5 Performance profiling with the TclX package

The TclX package has a set of profiling tools that collect data and generate reports based on that data. These tools were invaluable in figuring out what Tcl procedures and algorithms were using the most CPU cycles. I integrated the profiling package by creating a menu item which would pop up a display asking how many APS cycles I wanted to profile for. I went through several major repartitionings as I moved code from the Tcl side to the C side as the profiling highlighted hot spots. It is well worth the modest effort it takes to integrate profiling with your application.

3.6 Tcl Introspection

The introspection capabilities of Tcl provide a powerful method for troubleshooting. One of the difficulties of responding to user complaints about problems with a GUI system is figuring out how to later reproduce the situation the user was in. A technique used in APS is to provide the user with a menu entry that allows him to save the complete Tcl context by working through the entire set of global variables. By writing this out as sourceable Tcl code, it is often possible to restore enough context to reproduce the problem. In other Tcl/Tk programs I've developed, I've added a "panic" button that the user could push to automatically send me the complete crash dump plus any comments he wanted to add via e-mail. This "all-in-one" approach was not possible for APS because the configuration of the workstations does not include e-mail capabilities or an Internet connection.

3.7 Enhancing the Button widget

One of the requirements for APS was to simulate a technique that the operators use in the existing control room whenever the inputs to the alarm are being tested by the I&C maintenance staff. They have a special little tool that allows them to physically slide the alarm tile out of the control panel a centimeter or so. Then they know at a glance when they can simply ignore an alarm.

Pending the development of 3D holographic VDUs and a version of Tcl/Tk that supports them, we tried out a number of techniques for clearly indicating in two dimensions that an alarm tile was in test mode. We tried font style changes, button relief changes, contrasting border coloration, underlines and strikeouts, reverse video (exchanging the background and foreground colors), and text justification changes using the built-in capabilities of the Tk button widget and the Tk font command. While these changes were easy to try out interactively, none proved very satisfactory.

At the same time, we were trying to figure out an aesthetically pleasing way of letting the operator know that an alarm point was out of service or otherwise potentially not truly representing the state of the plant. In the Ovation process control system, the operators have many capabilities for manually taking points off scan, suppressing alarm limit checking, etc.. In addition, the system itself does a number of automatic data integrity checks to flag conditions such as sensor out of range and stale data from the data highway.

The ability to add extension code to Tcl/Tk includes the ability to add new widgets that can be used as if they were built in to the core. So I looked at creating a custom alarmtile widget starting from the existing button widget. The idea was to do as little work as and make as few changes as possible, knowing that the button widget was likely to evolve or at least have bug fixes applied over time. Further, I wanted the APS to be able to run with a standard button widget, even if the added features weren't supported.

My approach was to overload the "-underline" option to the button command. Perusal of the button widget code showed that if the index value was negative, no underlining would take place, and no bad side effects would occur. So I created a new widget starting with the tkButton.c/tkButton.h/tkUnixButton.c code that interpreted negative values supplied with the –underline option.

The visual effects that I added were a diagonal "slash" superimposed on the button/alarmtile rectangle (to indicate a computer-detected "alarm state not necessarily correct" condition) and an ellipse superimposed on the button/alarmtile rectangle (to indicate an alarm in test mode). Both visual effects were drawn using a stipple pattern and two colors (black and white) such that they always had at least reasonable contrast with the background color of the alarm tiles (which can be red, yellow, green, magenta, or gray depending on the alarm state). If you look closely at the center column of tiles in Figure 3 above, you can see what the diagonal slash and ellipse look like.

3.8 Internationalization with the msgcat package

The APS was developed for a Swedish customer, and because of this it was required to internationalize the menus and displays that the operator normally sees and uses (diagnostic error messages were not required to be in anything but English). The msgcat package made this very easy. I simply adopted the idiom of using

```
button .info.buttons.done    -text [language "Dismiss"] …
```

where language is defined as

```
proc language {text} {
        return [msgcat::mc $text]
}
```

To generate (and regenerate) the msgcat input files, I wrote a Perl script that found all the instances of the [language "XXX"] construct in the Tcl code and then wrote a msgcat format file containing lines of the form

```
msgcat::mcset sv "XXX" "(Sv):XXX"
```

The resulting file is then hand edited with the help of native Swedish speakers to provide idiomatically correct translations.

This scheme worked out well since when APS was placed into Swedish mode, it was immediately obvious whenever you saw a string prefixed with "(Sv):" that there were missing translations. It also made it easy to provide the customer with the complete set of strings that needed to be translated.

3.9 Interactive Font Experimentation

One of the nice features of Tcl/Tk is named fonts. With a named font, you essentially create an abstraction layer between the low level details of a font (size, weight, family, etc.) and the widgets that use the fonts. Once you create and use the named font in your widgets, you can later modify the font properties and instantly have the changes reflected in all of those widgets.

In truth, the usefulness of named fonts didn't make an impression on me, and in the initial versions of APS, I worked a lot harder than I had to in providing a method for experimenting with font properties. Eventually I discovered the font selection dialog example in the Welch book *Practical Programming In Tcl and Tk*, and adapted it to provide the ability to interactively experiment with the look and readability from a distance of various fonts. I should note that my work with this uncovered a bug (SourceForge ID 483832) when manipulating font properties on multiple X server applications. For my purposes, it was sufficient to use a single X server system for experimentation.

3.10 Automatic Testing with Tk Send

The APS is expected to run continuously for months at a time without failure. Such systems need to be thoroughly tested functionally (exercising all the code paths and features), under stress conditions (many alarms changing state at once), and over long periods of time (to check for memory leaks, operating system resource consumption, etc.). Running tests like these and analyzing the results is very boring, and is a perfect job for an automated test program.

APS configures itself as it starts up, and the test program needs to know the configuration so that it can properly manipulate the process inputs that will cause APS alarm state changes. Rather than duplicate the logic that reads the configuration files in the test program, I chose the following approach. When the test program starts, it sends a command to APS using the tk send command. The sent command invokes an APS procedure that uses Tcl's introspection features to dump all of the array and variable values to an output file in the form of Tcl statements. For example, if the APS has the array element box_geometry(sbox01), we write the statement *set box_geometry(sbox01) "+3123+1258"* to the output file. Once this is done, the test program then creates a new Tcl namespace and sources the output file into that namespace. The result is that the test program has access to all the configuration data it needs in a separate namespace that avoids any conflicts with variables and arrays in the test program's default namespace. Using this data, the completely generic test program can present

the user with APS instance-specific lists of available tests, test subsets, etc.

Another feature that facilitates testing is the power of the Tk event mechanism. Using the tk send command, we can simulate menu traversals, mouse button pushes and releases. When combined with the Ovation APIs methods for changing process point values, the ability to simulate user GUI actions, and the custom Tcl extensions that allow us to test the internal state of the APS, a complete closed loop test sequence was developed in a very short period of time.

The basic scheme is to dynamically write the test(s) (based on configuration values) in one step, and then execute the test(s) in the second step. Test results are displayed in real-time as well as being logged and categorized once it is known whether the test passed or failed. The entire APS can be exercised by simply starting the test sequence in the evening and checking the results in the morning. This testing tool has proven very useful for regression testing, as the addition of new features and capabilities to the APS has sometimes managed to break old features or expose latent bugs.

3.11 Interfacing with Netscape and Acroread

Tcl is good at controlling external programs with the exec command, and in APS we wanted to be able to control both Netscape and the acroread PDF reader for displaying HTML and PDF files respectively. Both acroread and Netscape provide a command line interface that allows you to specify the name of a file you want to display using an existing instance of acroread or Netscape. This avoids the problem of starting multiple instances if the user neglects (or chooses not) to close a previous one.

Using acroread in this manner was straightforward. One simply adds the "-useFrontEndProgram" option on the command line, and acroread uses the existing instance if any, or starts one if there isn't one. (Invoke acroread with the "–helpall" option to see all the available command line options and their syntaxes.). This means that the same basic command sequence

```
set command "-display $X_screen(acroread)  \
            useFrontEndProgram -name APS_acroread "
catch {eval exec $acroread_executable \
                        $command $PDF_path &} result
```

can be used every time.

Controlling Netscape was somewhat more complex. The Unix variant of Netscape provides the "-remote" option (see the URL http://wp.netscape.com/newsref/std/x-remote.html for details) to allow for remote control of a Netscape instance via X Windows features. There are two problems with this interface.. First, the "-remote" option fails unless an instance of Netscape is already running on the X display. This means that you cannot just use the same command line recipe each time you want to display an HTML file. You must distinguish between the first time and all subsequent times. In APS, I bypass this problem by simply starting Netscape as APS starts up, taking care to clean up the Netscape lock file if necessary. I should note in passing that checking for the lock file was not as simple as using the Tcl command "file exists "$env(HOME)/.Netscape/lock". Netscape creates the lock file as a symbolic link that looks like

/home/fitchk/.Netscape/lock -> 168.92.189.94:616

which encodes the IP address and the Unix process id. Since the encoded string is not the name of a file, the file does not "exist" as far as Tcl is concerned. Instead, I had to use the file readlink command to see if the lock file was there or not, and to determine which host created the lock file

At this point, if there is a lock file, we can see if the Netscape instance was created by the host running APS. If it was, we use the process ID to kill it before starting a new Netscape instance, on the theory that the computer crashed or a previous instance of APS was abruptly killed without have the chance to clean up. If the Netscape instance was created by another host, we simply announce that fact and leave it alone.

Note that the very useful (and not immediately obvious) idiom for finding out one's own IP address with the Tcl proc

```
proc my_ip_address {} {
      set me [socket -server garbage_word -myaddr \
            [info hostname] 0]
      set ip [lindex [fconfigure $me -sockname] 0]
      close $me
      return $ip
}
```

came in very handy in this application. I discovered this while reading the comp.lang.tcl newsgroup one day, and it is a much cleaner approach than my original scheme of using ypmatch output or searching through /etc/hosts to make the hostname – IP address association.

A second problem was a side effect of the thousands of widgets that APS puts up on the screen. When Netscape is started with the "–remote" argument and without the "–id window" argument, it uses the XQueryTree function to search for a Netscape window on the display. But when there are thousands of windows, it often takes ten to fifteen seconds to find the right window ID. That meant that the first time an operator asked for an HTML file, it took an unacceptably long and unpredictable time to display it. The "-id window" argument to Netscape allows the user to specify the window running the target Netscape instance directly, so that no search is required. The question then becomes how to find out what the window id is, as this will be different every time you start up Netscape.

By experimentation, I determined that Netscape names the window based on the title of the HTML document. I used this in the startup sequence by synthesizing a simple HTML file with a known title and then telling Netscape to display that file when initially started. Then I use the Tcl sequence shown below to start up Netscape and check at repeated intervals to extract the window ID using xwininfo.

Note that in practice, these contortions should only be needed in a system with widget counts in the many hundreds or more.

```
set URL "$local_file_url_prefix$magic_path"

#  request the magic page be displayed on the netscape
#  running on my display

catch {exec $netscape_browser –display $X_screen(netscape) \
                        $URL -iconic &} netscape_pid
```

```
set attempts 0
while {$attempts < 5} {
        incr attempts
        after 2000
        set status [catch {[exec xwininfo -display
                $X_screen(netscape)  -name "Netscape:
                $magic_title"]} result]
        if {[regexp {(0x[0-9a-fA-F]+)} $result netscape_Xid]} {
                debug_message "xwininfo success"
                break
        } else {
                debug_message "xwininfo failure"
        set netscape_Xid 0xdeadbeef
}
}

if {$netscape_Xid == 0xdeadbeef} {
        log_message "Netscape inoperative ... continuing"
}
```

In both cases, we record the Unix process ID (pid) in a Tcl list named "grim_reaper_list" to use to kill off any instances of auxiliary processes (e.g. Netscape or acroread) that were started by APS. When APS exits cleanly, we issue Unix kill commands on every pid in the grim_reaper_list.

4 SUMMARY OF MY EXPERIENCES

The use of Tcl/Tk as the basis for the APS was a major factor in what Westinghouse and the Ringhals customer both agree has been a success to date. (The system is not yet installed and running, so the final declaration of success must wait.) The implementation of APS came in significantly under the projected budget. The ease of making changes and experimenting with the look and feel of the GUI meant that we could try stuff without a major effort. This is of great importance both in involving the customer in the design process and in winnowing out the ideas that sound good on paper but end up working poorly in practice.

From my perspective, working with Tcl/Tk is a lot of fun. I enjoy using powerful programs like Tcl/Tk that allow you to focus on what you want to achieve without having to deal with the myriad details of making things like scrollable text windows with embedded widgets appear on the screen.

Here are some things I'd like to see incorporated into Tcl/Tk.

- A better way of keeping track of torn-off menus. Currently, a torn off menu becomes an "orphan" when torn off (it is assigned a widget name of .tearoff#, where # is some number) and there's no easy way to later find the proper widget name to close the menu when the window that spawned it closes. Perhaps something as simple as taking the name of the original menu, replacing all of the widget tree "." separators with "_" characters, and then naming the tearoff as ".tearoff_<original_with_underscores>" would work.

- Some form of widget with "transparent" properties that could be superimposed on another widget. This feature would have helped me avoid the development of the alarmtile widget described above. I'm not sure how difficult this would be, especially for Windows or Mac.

- A way of having Tcl procedures implicitly declare as global variables that it cannot find defined locally in the procedure. The most frequent Tcl coding error I make is to forget to add the global statement in every procedure.

And finally, I'd like to thank all the people who have contributed code or ideas to Tcl/Tk over the years. The APS was built upon the foundation you laid.

5 REFERENCES

[1] IEC 1226, "The Classification of Instrumentation and Control Systems Important to Safety for Nuclear Power Plants", 1993

[2] URL for info about the BARCO EOS:

http://www.barco.com/projection_systems/products/product.asp?element=342

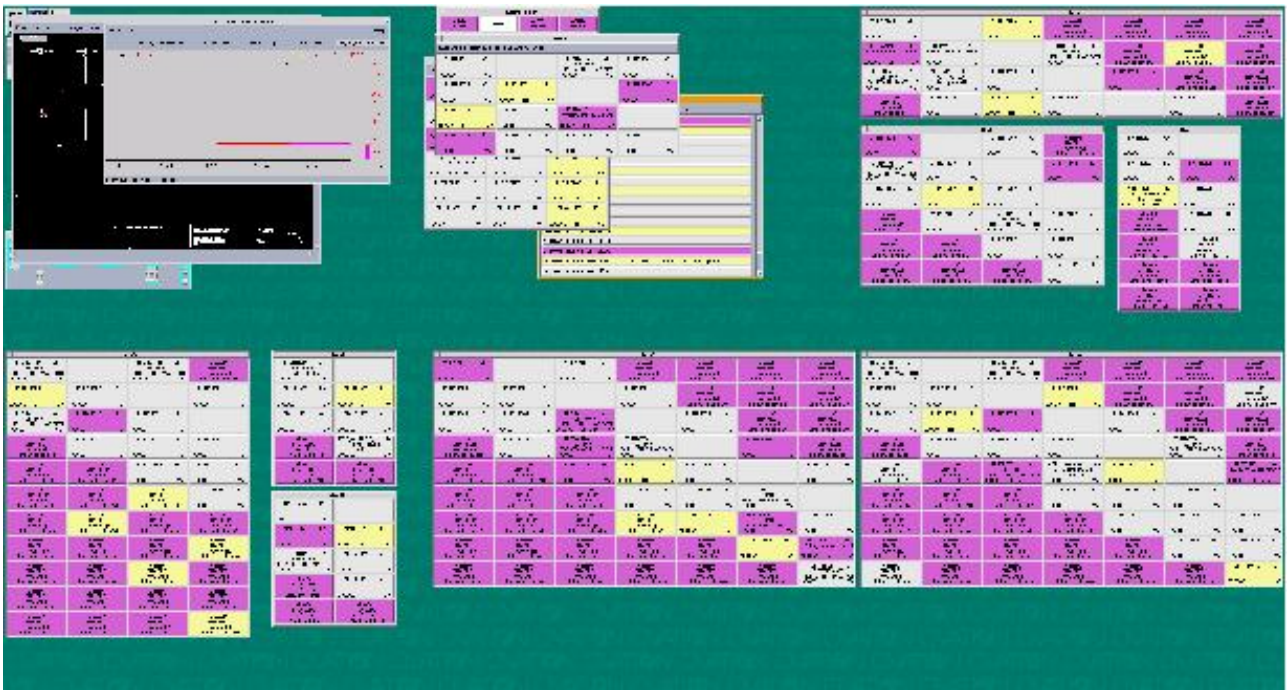

Figure 10 – The APS Logical Screen - 3840x2048 Pixels