Astronomical Data Analysis Software and Systems IV
ASP Conference Series, Vol. 77, 1995
R. A. Shaw, H. E. Payne, and J. J. E. Hayes, eds.

Tcl/Tk with DRAMA - A Natural for Building User Interfaces to Instrumentation Systems?

T. J. Farrell, J. A. Bailey, and K. Shortridge

Anglo-Australian Observatory, P.O. Box 296, Epping N.S.W. 2121 Australia

Abstract. DRAMA allows you to build distributed real-time systems consisting of a set of event-driven tasks. The Tk windowing system is also event-driven—like most windowing systems. The similarities between the two have allowed them to be merged extremely effectively, providing an almost seamless interface between the two. The control system for the AAO's 2dF project has instrument control tasks written using DRAMA and running on VMS, UNIX, and VxWorks systems. These are tested and controlled by higher-level tasks whose user interfaces are provided by Tcl/Tk and which communicate with the rest of the system using purpose-built Tcl commands that invoke the DRAMA system routines. This gives the critical instrument control tasks the fixed nature and reliability that comes from a compiled, linked system, while allowing a very flexible user interface and test system that takes advantage of the ease of modification and the flexibility of an interactive language like Tcl/Tk.

1. DRAMA - A Quick Introduction

The AAO's Two degree Field (2dF) project, involving two robotic positioners manipulating two sets of 400 optical fibers and multiple detector systems, will be the AAO's most complex user instrument. The DRAMA system that has been developed for its control is a portable environment designed for writing instrumentation software. It has similarities to the Starlink ADAM environment (Kelly 1992), but is written entirely in C and has been ported to VMS, various flavors of UNIX and the VxWorks real-time kernel.

Its basic unit is the *Task*. A DRAMA Task is normally implemented as a separate process within a multi-process operating system. A Task may send or receive messages of various types. The most fundamental message type is the "OBEY" message. An associated name specifies the name of an "Action" that the task will perform. The simplest DRAMA task will set up relationships between "Action" names and C routines. It then enters a loop where messages are received and dispatched to the routine that is the current "handler" for that action.

A DRAMA task may have parameters associated with it which may be read or written by other tasks. So we have "GET" and "SET" messages which get and set the value of a named parameter. We also have "MONITOR" messages which allow a task to be notified if the value of a parameter in another task changes. Task parameters may be hierarchical structures of considerable complexity (although they are often simple scalar values) and a DRAMA message may contain such a structure, so this provides a very flexible way of transferring information around a DRAMA system.

DRAMA is well suited to both the lower level control of individual instruments and to the coordination of the overall system. What was lacking a year ago was a clear direction for the user interface. Although we had made it possible for a DRAMA task to include an X-based interface, and had some tasks written with Motif interfaces, it was clear that this was not an easy system with which to write flexible or experimental user interfaces.

2. DRAMA and Tcl/Tk

Tcl (Ousterhout 1994) is an easily extensible scripting language that is becoming very popular. Tk is an extension to Tcl that supports building graphical user interfaces under X-Windows. Initially we considered using Tcl to provide a scripting language for automatic test procedures for DRAMA. It turned out that because both Tk and DRAMA are event-oriented systems, the basic DRAMA messaging operations fit very well into the Tcl/Tk approach. For example, to allow Tcl/Tk applications to send the message types mentioned above, we implemented new Tcl commands named `obey`, `pget`, `pset` and `monitor`. (get and set are already defined by Tcl.) These are all similar. Let us consider the `obey` command.

The basic form of the `obey` command is

```
obey task action [args] [options]
```

Where `task` is the name of the DRAMA task to send the message to and `action` is the action name. `Args` holds the optional arguments to the action. Various options are available including

-success *command*. This specifies a Tcl command to be executed when and if the obey completes successfully.

-error *command*. This specifies a Tcl command to be executed if the obey completes with an error.

As a very simple example let us consider a one button Tcl/Tk application that sends a single DRAMA obey message when the button is pressed. The Tcl code is

```
button .a -text "my button" -command {obey CAMERA EXPOSE  \
        -success { puts "Expose action completed ok";# } \
        -error   { puts "Expose action completed with error";# }}
pack .a
```

The Tk command `button` creates a button widget and `pack` makes it appear. The label of the button is specified with `-text`. A Tcl command to be executed when the button is pressed is specified with `-command`. In this case, it invokes the action `EXPOSE` in a task named `CAMERA`. Note that this all happens asynchronously. The process running the Tcl script continues running, and the user interface remains responsive, once the obey command is sent. Eventually, the message from the camera task indicating either success or failure will arrive and the appropriate action will be taken by the Tcl script.

It is possible to replace the argument to the `button`'s `-command` option and `obey`'s `-success` and `-error` options with the names of Tcl procedures to be invoked, allowing operations of any required complexity to be performed. In this style of application Tk events such as button presses are seen as events that initiate DRAMA messages. The responses to the DRAMA messages can then trigger any required changes in the user interface. A user interface may add C code to implement additional Tcl commands. An example would be code to perform mean to apparent place conversions, which can be done in Tcl, but for which C routines already exist.

3. Monitor Messages

Monitor messages have proven particularly useful. The monitor mechanism allows one task to "express an interest" in a parameter of another task. When this parameter changes, the interested task receives a message identifying the parameter and containing its new value. This solves a common problem quite neatly. Often, a low level task, controlling some part of an instrument such as a spectrograph, will have information such as grating angles, filter positions, etc. that is of interest to other parts of the system such as the user interface. However, the modularity of the system is broken, awkwardly, if such a task has to make assumptions about the higher levels of the system. You would not want to write a spectrograph control task with explicit code that always sent a new filter position to a task called "USER_INTERFACE", since you may be running with a completely different user interface task.

It would be possible for the spectrograph task to have an mechanism coded into it whereby it could be sent the name of a task which it was to notify whenever a filter position changed. This solves the modularity problem, but requires a lot of explicit code in the low level task. Alternatively, the user interface could poll the filter value, but this is inelegant.

Under DRAMA the combination of the monitor facility with the very flexible task parameters normally supported allows a neat solution to the problem that is event-driven and which requires no specific code in the low-level task. The user-interface code that "expresses an interest" in the low-level task's parameter is just a few lines of Tcl. The user interface can even direct that an image parameter maintained by a camera task be monitored by an image display task, with no specific code required in either the camera task or the image display task.

This could also be used to help implement remote observing. We envisage a scheme where two copies of a user interface are run, one local and one remote. The lower level DRAMA control software would be run locally and commands sent from either the local or remote user interface using normal DRAMA messages. Monitor messages would allow both user interfaces to be kept up to date with what the other was doing to the system using a minimum of network traffic.

4. 2dF Commissioning

The combination of DRAMA and Tcl/Tk let us put together simple user interfaces for testing individual components of the 2dF system. We were also able to produce high quality overall system interfaces, controlling multiple low-level tasks.

During the initial commissioning run for 2dF, we found ourselves struggling with a VME system that would not reboot cleanly. This discouraged us from making any changes which might require a restart of the system. The ability to dynamically change a Tcl/Tk program and execute simple DRAMA commands proved invaluable. When we devised fixes or improvements to the user interface, it was easy to load and test them without having to risk rebooting the faulty VME system.

5. Conclusion

DRAMA and Tcl/Tk seem to work together extremely well. The original DRAMA design provided a good environment for writing distributed real-time systems running on a disparate set of machines. With Tcl/Tk it is now easy to produce powerful and flexible user interfaces that can take advantage of the underlying DRAMA system.

References

Kelly, B. D. 1992, ADAM—Guide to Writing Instrumentation Tasks, Starlink User Note 134

Ousterhout, J. K. 1994, Tcl and the Tk Toolkit (Reading, Addison-Wesley)