NBC's GEnesis Broadcast Automation System: From Prototype to Production

Stephen J. Angelovich
NBC Broadcast and Network Operations
Kevin B. Kenny and Brion D. Sarachan
GE Corporate Research & Development

NBC's GEnesis Broadcast Automation System: From Prototype to Production

Stephen J. Angelovich
NBC Broadcast and Network Operations

Kevin B. Kenny
Brion D. Sarachan
GE Corporate Research & Development
`kennykb@crd.ge.com`

Abstract. GEnesis is a system in use at the NBC television network for automating the composition and distribution of video. It works in a mission critical environment; a system failure could potentially result in a substantial loss of revenue for the network. Tcl/Tk has been an integral part of the operator interface and data handling portions of the GEnesis system from the earliest stages of prototyping. We originally planned to replace the system prototype based on Tcl/Tk with a production system built in a compiled, object-oriented language and using commercial component software. After the prototype phase was completed, the developers and management together decided to keep numerous system components in Tcl, while migrating some complex and performance-critical functions from Tcl to a C++ message passing architecture. This paper discusses that decision and presents our experience with converting the prototype into a fully functional system.

1. Introduction

GEnesis is an upgrade to the NBC television network to support the requirement for digital video and to increase the network's capacity from ten simultaneous streams of video to forty. It is an ambitious control system; it includes roughly 400 computer-controlled devices for processing, storing, and routing video. Among its components are some sixty devices for video storage and processing that include RAID arrays totaling several tens of terabytes of disk space, satellite uplink/downlink controls at over two hundred stations, and high-bandwidth digital video routers to interconnect the devices. The system does not run production studios nor transmitters at the local stations, but handles all the tasks needed in between: video storage and playback, combining video segments into an integrated stream, adding special effects, doing voice-overs, and managing the satellite distribution system for over 200 NBC affiliate stations. An overview of the system appears in Figure 1.

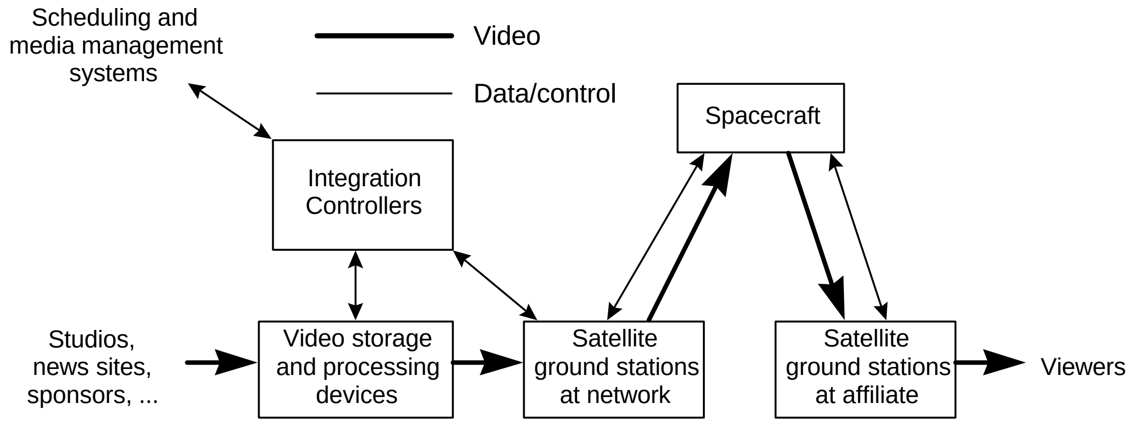

Scheduling and media management systems
Video
Data/control
Spacecraft
Integration Controllers
Studios, news sites, sponsors, ...
Video storage and processing devices
Satellite ground stations at network
Satellite ground stations at affiliate
Viewers

Figure 1. GEnesis system overview

Tcl/Tk was chosen over four years ago to build prototypes for several user interfaces in the GEnesis system. The idea at the time was to use Tcl/Tk to build a working model, test new ideas, and expose a variety of proposed user interfaces to the system operators. At the time, there were no plans to preserve any of the prototype into the deployed system.

As we described in the 1995 Tcl/Tk Workshop [SSZ95], Tcl was well suited forprototyping the GEnesis application. After the publication of our previous paper, prototype development continued to proceed rapidly. We completed 1995 with a comprehensive GEnesis prototype, including simulators for digital video servers and routers, all implemented using Tcl/Tk, with database access using `sybtcl` [Poin97] and interprocess communication using `Tcl-DP` [SRY93].

When we started building the production system, we decided to take the radical step of keeping Tcl/Tk when building the Integration Controllers — the computers that accept the network schedule from database systems upstream, provide the user interface to the network operators, and deliver commands to the video devices. Performance-sensitive components, and components that required access to third-party interfaces, were migrated to C++. The user interface continued to be built with Tcl/Tk. The C++ code supported extensive accessor functions that allowed it to be configured and commanded from Tcl. This design had several advantages. It allowed extensive configurability: the same executable is now used to build six different types of operator station, plus a wide array of plug-in test harnesses. It allowed us continued access to the Tk text and canvas widgets; no other user interface toolkit seems to have anything nearly as effective for customizable user interfaces. It allowed us to have a working version of the system at all times; there was no early period when the system was “still under development” with nothing available to demonstrate. Perhaps most important, Tcl’s embeddable nature allowed us to develop interfaces to the actual devices even though they came from several vendors and had widely varied interface conventions.

2. From prototype to deployed system

We continued development with a natural iteration among prototype development, use case definition, requirements understanding, and architecture development. The development was much the same as the "star" approach described in [HH89]. We felt that this development process was ideal for a project like GEnesis, in which the resulting system was to be much different from the system being replaced; understanding and defining the requirements was part of the ongoing development process. The requirements could not possibly have been laid out at the outset according to the traditional "waterfall" development process. The “star” approach has allowed us to converge on the requirements in close collaboration with our colleagues at NBC.

One of the keys to our success in this project was to migrate gracefully from prototype to deployed system. We have had a working system at all times since we began in late 1994, with increased levels of functionality, reliability, and performance. We have never allowed ourselves the luxury of a "big bang" that would break the system for an extended period. In the later phases of development, we have constructed daily builds of the software, complete with automated installation scripts. Nearly every build has been tested immediately at NBC.

As stated above, our initial prototypes were implemented entirely using Tcl. This section summarizes some of the key steps as we migrated to a highly reliable, production quality software system.

2.1 Message passing architecture

The Integration Controller supports a complex set of functions, which include such things as simulating the device-level execution of the network television schedule, and reconciling the simulation against status returned by the actual devices, to determine whether execution was correct. We originally implemented much of this logic in Tcl. This exercise was highly valuable for prototyping the logic and understanding the requirements.

The complexity of this implementation in a scripting language did eventually become unwieldy. In particular, we fell victim to namespace pollution. Global arrays (representing, for instance, all events using a given device in time sequence, all events referring to a particular video clip, and so on) proliferated, and maintaining a dictionary of their names became unwieldy. Moreover, the arrival of each event resulted in the execution of many thousands of lines of Tcl code to maintain the data structures. The pure Tcl implementation had trouble keeping up with its workload.

We developed a C++ object-oriented architecture, known as "NetSys," to provide a maintainable and extensible infrastructure for the Integration Controller's processing. (NetSys itself has many interesting aspects to its design, which are beyond the scope of this paper.)

2.2 Extensible Tcl event loop

The Integration Controller is an event-driven application that receives data, control signals, and device status through a variety of communication protocols, including TCP and UDP sockets and proprietary protocols. We were able to make appropriate extensions to and callbacks from the Tcl event loop to support all of these interfaces, in addition to the user interface events that it already supported.

Integrating additional event sources into the Tcl event loop proved to be extremely simple. Only two areas were really cause for concern. The first of these was the fact that non-blocking interfaces are unnatural on the Windows platform, where multithreaded applications are the rule rather than the exception. Some of the third-party interfaces that we must use, therefore, freeze the event loop for longer than we would like while they are performing their tasks. We deal with this situation by the expedient of putting the code that uses these interfaces into separate processes, and communicating with them using sockets. If the event loop stops responding temporarily, our message queues hold the traffic for these processes until it starts again. We may revisit this decision when the thread-safe Tcl core becomes available in release 8.1.

The second cause for concern is the stability of the interfaces. It seems that any C code that uses the Tcl event management primitives breaks with every release of Tcl, since the protocols change so rapidly. Tracking these changes over the life of the project was a major headache, and some of the changes seemed gratuitous. We hope that the multi-threaded notifier will represent the last round of major incompatible changes.

2.3 Uniform interfaces

On several occasions we replaced prototype functionality while keeping the interfaces constant. This greatly facilitated our goal of always having a working system. For example, the early prototype code often used Tcl associative arrays as data structures. When moving to C++, we have often provided bi-directional mappings from C++ objects to Tcl associative arrays, allowing for interoperability between the Tcl and C++ implementations.

The NetSys library provides a uniform messaging interface to socket connections, database access, device drivers, user interface displays, and internal processing. By using Tcl interfaces like the ones presented here, functionality can be organized in different configurations through Tcl scripts which configure the NetSys message handlers.

The mapping between objects and associative arrays turns out to be surprisingly easy to do. Each C++ class has a few stylized static methods that hook it into Tcl. The `constructFromTclArray` method (Figure 2) extracts the values from the Tcl array and invokes the C++ constructor.[*] It then installs the newly-created object into the Tcl command namespace.

The `tclCommandDeleteFunction` method (Figure 3) is a trivial connection to the destructor.

The `tclCommandFunction` method provides whatever method calls are needed from Tcl. One interesting technique that we use frequently is to pass a C++ object by reference to a method in another C++ object. The Tcl interface is to pass the name of the Tcl command that represents the object. The `Tcl_GetCommandInfo` library function is used to validate that the string is the name of an object of the correct type, by examining the command function pointer (Figure 4).

When filling an associative array with the contents of a C++ object, we have usually found it more convenient to create a list of alternating keywords and values, return that list to Tcl, and use the `[array set]` command to install the values in the Tcl array. This trick meant that the C++ code did not need to be prepared to deal with possible error status returned by `Tcl_SetVar2`.

2.4 User interface evolution

Several of our prototype user interface screens were shown in our earlier paper [SSZ95]. One result of our “star” development process was to continually refine these screens based on user feedback. Tk provided an ideal tool for easy changes to the user interface. We wholeheartedly agree with the sentiments expressed by Brian Kernighan in [KERN95]; the Tk text and canvas widgets provide power and flexibility at least as good as anything else on the market.

One specific change that the NBC users requested early was to replace our early “busy” screens with a simpler layout, and provide a rich assortment of application views as “notebook tabs,” as is familiar in many modern Windows-based applications. It might not have been inordinately difficult to implement this look and feel using the canvas widget (as Harrison and McLennan do in [HM98]), but it would have been time-consuming. Instead, we took advantage of the Tcl community and adopted Ioi Kim Lam's Tix widget set, which provided us with a notebook widget off-the-shelf.

[*] All of the illustrative code is targeted to Tcl 7.6, which is now obsolete. The GEnesis project continues to use it because it works adequately well, and the benefits to be gained from moving forward to the Tcl 8 object APIs are not yet worth the effort of converting the C++ code.

```
int
myClass::constructFromTclArray (Tcl_Interp* interp,
                                char* arrayName)
{
    char* string;        // working storage
    int parameter1;      // constructor parameters
    char* parameter2;
    if (string = Tcl_GetVar2 (interp, arrayName,
                              “parameter1”, TCL_LEAVE_ERR_MSG) == NULL)
      return TCL_ERROR;
    if (Tcl_GetInt (interp, string, &parameter1)) != TCL_OK)
      return TCL_ERROR;
    if (parameter2 = Tcl_GetVar2 (interp, arrayName, “parameter2”,
                                  TCL_LEAVE_ERR_MSG) == NULL)
      return TCL_ERROR;
    myClass* newObject = myClass (parameter1, parameter2);
    char instName[24];
    sprintf (instName, “myClass_%p”, (void*) &newObject);
    Tcl_CreateCommand (interp,  instName, myClass::tclCommandFunction,
                       (ClientData) newObject,
                        myClass::tclCommandDeleteFunction);
    Tcl_SetResult (interp, instName, TCL_VOLATILE);
    return TCL_OK;
}
```

Figure 2. Constructing a C++ object from a Tcl array

```
static void
myClass::tclCommandDeleteFunction (ClientData clientData)
{
    delete (myClass*) clientData;
}
```

Figure 3. Destroying a C++ object from Tcl

```
Tcl_CommandInfo info;
if (Tcl_GetCommandInfo (interp, commandName, &info) == 0) {
    Tcl_AppendResult (interp, name, ": no such command", (char*) NULL);
    return TCL_ERROR;
  }
  if (info.proc != &DesiredClass::TclCommandFunction) {
    Tcl_AppendResult (interp, name,
                      " is not an instance of DesiredClass",
                      (char*) NULL);
    return NULL;
  }
  return (NodeBaseClass*) info.clientData;
```

Figure 4. Type-checking an object passed by name from Tcl

2.5 Port from Solaris to Windows NT

Another major benefit we derived from Tcl/Tk was portability. Two years into the project, NBC decided to change computer platforms from Solaris to Windows NT. The porting effort was minor, thanks to the fact that Sun's first Windows port was released just in time. We have evolved a team development environment that uses Windows-based tools (Microsoft Developer's Studio, MKS Source Integrity, Purify, pcAnywhere, and other tools) and suits our purposes well. Had we not chosen Tcl/Tk early on (the original proposal called for implementation with OpenWindows and Motif), the unexpected port to Windows could easily have been a showstopper.

3. Using Tcl in critical systems

The GEnesis system had tight requirements in a number of critical areas. It required very high *reliability:* system failures must be few or nonexistent (the eventual system is targeted to have less than 20 minutes of unscheduled outage in a year's operation). It has several tight *performance* requirements: it has a complex graphical user interface that may have several dozen objects updated in a second. It also has a *longevity* requirement: the operator's workstation cannot be interrupted more than about once a week, and the applications have to be able to run that long without restarting. After a small number of issues were resolved, as summarized below, Tcl/Tk together with custom extensions provided a robust platform which supported the reliability, performance, and longevity requirements of GEnesis.

3.1 Reliability

Our experience has been that Tcl/Tk has presented few reliability problems. In the course of deploying the system, we discovered several bugs in the Tcl/Tk implementation that resulted in program crashes. In all cases, the Sun staff were able to find and correct the problems in short order. In the last several months of operation (since the bugs that we encountered were fixed or worked around), no system failures have occurred that can be ascribed to problems with the Tcl/Tk core.

3.2 Performance

As we have already discussed, the Tcl interpreter was too slow for various operations in the system that involved complex data structures. We reworked these operations in C++. Ultimately, the parts of the system dealing with device control and data management evolved to being built entirely from C++ objects, with Tcl used as a configuration language to string these objects together.

Performance of the graphical user interface has been a more difficult problem; we ran into several entirely unexpected performance problems, some of which proved hard to characterize. The first of these is simply the vast amount of memory allocation activity that Tcl requires (another culprit here is the Rogue Wave libraries [RW96], which we use extensively). Our initial development environment used the "debugging" versions of `malloc` and `free`, which cleared memory and had basic integrity checking. When we replaced these with the non-debugging versions and instrumented the code, we saw a 50% performance gain for the common operations of starting and stopping video clips. We have not resorted to the non-debugging libraries in practice, feeling that the additional checking gives us a safety net. Nevertheless, the temptation is there, and we may succumb to it at some time in the future.

A second user-interface performance issue is the Tk console on Windows NT, which is simply too slow to use for more than a tiny volume of output. In addition, its performance appears to degrade rapidly as more text is added to it — a console with a few hundred lines of text in it is noticeably sluggish. Using the console as a message log is a longevity issue as well, since printing to the console consumes memory rapidly, using the better part of a kilobyte of heap space to display a one-line message. One part of our testing procedure is to inspect the Tcl code rigorously to make sure that all `puts` directives to the console are removed. In addition, to cover any console output that may escape this net, we have added to our initialization a script that looks like Figure 5.

```
proc keepConsoleClean {} {
  console eval {
    .console delete 1.0 end-100l
  }
  after 15000 keepConsoleClean
}
keepConsoleClean
```

Figure 5. Script to limit the console display

Another performance issue that recurred several times in the course of development was the management of tags in the text widget. One central part of the GEnesis system is the display of lists: lists of video to play, list of files to transfer among the playback devices, lists of available video clips, and so on. These lists are displayed in columns in the text widget, and the operator is provided with a quick-and-dirty query mechanism in which a double-click in any item highlights all lines having the same value in the item's column. This mechanism allows very quick answers to questions like, "what events use playback device CDX-01?” or “at what times are we scheduled to run the latest *Third Rock* promo?”

Our original design for the system used multiple text tags on each line, one for each field value, and did the highlighting by changing the display attributes of those tags. When we tried this scheme with hundreds or thousands of lines, however, we found that it was totally unworkable. The double click sometimes required many seconds to produce results, and the entire user interface was frozen for that length of time. To avoid this issue, we developed a complex scheme of tag management, which is of sufficient interest that we present it separately in Section 4.

3.3 Longevity

Achieving the longevity needed for a system in 24×7 operation was also a challenge. The problem here was a variety of resource leaks. The initial prototype for the system [SSZ95], which was built in Tk 3.6 on a Unix system, was awful in this regard, because of the way that Xlib leaked resource identifiers, and often crashed within a few hours. We are grateful to John Ousterhout for having fixed this problem in release 4.0.) The problem of memory leaks on the X11 platform is still intractable (we have found that X11 servers from several vendors need to be restarted every week or so); fortunately, it appears to be less of an issue on Windows NT, where we routinely run Integration Controllers for weeks at a time.

Of course, we continually have to scrub our C++ code for leaks. One recurring issue with Tk results in memory leaks: the `Tk_Uid` data type. This data type is used internally to Tk for making single copies of the names of objects: widget names, text and canvas tag names, and so on. The copies are kept in a hash table, and pointers to the strings may be compared with a simple comparison of pointers rather than a full string comparison.

Unfortunately, the `Tk_Uid` scheme assumes that there will be some small fixed set of names. If any tags or widgets are assigned names based on their content or based on an incrementing sequence number, the corresponding `Tk_Uid` objects are created and never recycled, resulting in a constant memory drain. This problem hit us unexpectedly several times. The most difficult was in the area of text and canvas tag management. We wished to tag lines of text and canvas items so that we could rapidly find the items displaying particular data. Since some of the data are never reused (in particular, the start time of video events continuously advances), any obvious tag scheme caused `Tk_Uid` objects to be created endlessly. We eventually developed a complicated tag management scheme to deal with this problem.

We found ourselves unable to resolve some of the issues of managing `Tk_Uid` objects, and dealt with them by negotiating away requirements. In particular, the customer once requested that the title bar of the main user interface window show a summary status line, with the current time and various parameters relating to the load on the system. We spent half an hour or so coding up this functionality in Tcl, only to discover that the `wm title` command creates a `Tk_Uid` object for the window title. This leak consumed memory at a rapid enough rate to crash the application after only a few hours. When we analyzed the problem, we decided that the functionality was not worth the amount of time that it would take to fix the problem in the Tcl core. (To the best of our knowledge, this is the only Tcl/Tk bug that we encountered but didn’t actually fix.)

4. Tk tag management

In order to address the problems of text and canvas tag management, we were forced to develop our own tag maintenance system that layered on top of Tk’s system. The system was designed to meet the following constraints:

- Inserting and deleting tagged items must be fast.
- Every item must have a unique tag, so that no tag will span a large fraction of the text widget or canvas display list.
- Tags must be drawn from a limited set of identifiers that are recycled to avoid leaking `Tk_Uid` objects.

The solution that we chose was to represent the tag structure in a family of global variables. Each of these has a window path name as part of the array name; they are brought into scope with the `upvar` command:

```
upvar #0 next_unused_tag$w \
                next_unused_tag
upvar #0 free_tags$w free_tags
upvar #0 id_for_tag$w id_for_tag
upvar #0 tag_for_id$w tag_for_id
# ... and so on ...
```

When inserting an item into a widget, the first thing that the code must do is locate an available tag to label the item. It does so with code like the following:

```
set searchKey \
  [array startsearch free_tags]
if {[array anymore \
        free_tags $searchKey]} {
  set tag [array nextelement \
            free_tags $searchKey]
} else {
  set tag [incr next_unused_tag]
}
array donesearch \
        free_tags $searchKey
```

Similarly, when deleting an item, its unique tag must be recycled:

```
set free_tags($tag) {}
```

Note that the set of recycled tags is maintained as the subscripts of a Tcl array (whose values are immaterial), and not as a Tcl list. We do this to avoid yet another performance problem. If the system has been under heavy load and subsequently is unloaded, there may be a great number of recycled tags. The code that removes one tag from the list:

```
set freeTagList \
  [lrange $freeTagList 1 end]
```

would have to copy the entire list. Recycling N tags would take $O(N^2)$ CPU time. This behavior surprised the original programmer, whose experience with Lisp suggested that there should be a constant-time operation analogous to Lisp's `cdr` function.

The `id_for_tag` array is used to map the unique tags back to object names in our C++ system, and the `tag_for_id` array maps the object names to the tags. These arrays are updated whenever items are created and destroyed.

Changing the appearance of an item or group of items, and responding to event bindings, requires executing `foreach` loops to locate the affected objects. Surprisingly, this scheme is much faster than the naïve scheme of tagging each item with its attributes, in addition to avoiding the problem of `Tk_Uid` leaks.

5. Troubleshooting resource management problems

While working on performance and longevity issues, the Genesis team used and developed a number of tools to track down specific CPU "hot spots" and memory management problems.

5.1 CPU performance measurements

It is virtually impossible to make headway with performance problems without profiling the code to find where it is spending its time. Tcl code, alas, confuses most profiling systems. When addressing performance issues, we tried profiling at the C/C++ level using tools like the profiler included with Microsoft Visual C++ and Pure Quantify. These tools were invaluable in isolating certain problems in our C++ code, but were virtually useless for telling where a Tcl program spends its time. (We already knew that it spends its time in `Tcl_Eval`, thank you!)

The profiler supplied with NeoSoft's Extended Tcl [LeDi89] was more effective, since it showed the time consumption based on the Tcl call tree. It showed only the time spent locally to a procedure, however, rather than the time spent in the procedure and the ones it calls. Given that Tcl programs are usually structured as many small procedures with complicated interrelationships, isolating performance issues from this output was also not easy.

Finally, in desperation, we implemented our own data reduction atop the NeoSoft profiler. Our system (Figure 6) uses the hierarchical list widget in Tix [LAM95] to display the call tree, and shows the real and CPU time spent in the procedure, and in the procedure plus descendants, next to the procedure's entry. Browsing through the output in this form makes the trouble spots obvious. (It would still be nice to tie them to source file and line number. We hope that the forthcoming TclPro product will finally make it possible.)

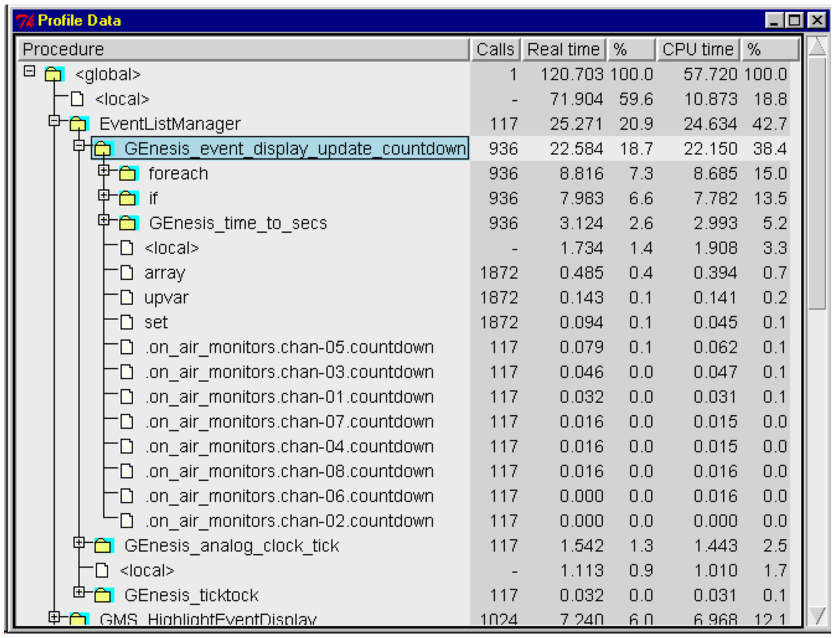
5.2 Resource leak identification

We use the Purify system extensively to track memory leaks. Unfortunately, we are handicapped by the extensive volume of the output that it produces. Not only Tcl, but also several other third-party software libraries that we use allocate memory at initialization time that is freed only by process exit rather than explicit calls to `free`. This memory is reported as "in use on exit" or "leaked on exit," exactly as if it were really leaked. We get around this problem by running the system for long periods with synthetic workloads, and then scanning the output of Purify for leaks involving large numbers of blocks.

In addition, we have developed a few little Tcl procedures that monitor the command and variable namespaces and report changes. These can be used to identify variables, array elements, and commands that are created and never deleted.

In our experience, the most difficult leaks involve the following areas:

- Traces established on nonexistent variables.
- Text and canvas tags that do not apply to any items in the widget.



Procedure	Calls	Real time	%	CPU time	%
<global>	1	120.703	100.0	57.720	100.0
<local>	-	71.904	59.6	10.873	18.8
EventListManager	117	25.271	20.9	24.634	42.7
GEnesis_event_display_update_countdown	936	22.584	18.7	22.150	38.4
foreach	936	8.816	7.3	8.685	15.0
if	936	7.983	6.6	7.782	13.5
GEnesis_time_to_secs	936	3.124	2.6	2.993	5.2
<local>	-	1.734	1.4	1.908	3.3
array	1872	0.485	0.4	0.394	0.7
upvar	1872	0.143	0.1	0.141	0.2
set	1872	0.094	0.1	0.045	0.1
.on_air_monitors.chan-05.countdown	117	0.079	0.1	0.062	0.1
.on_air_monitors.chan-03.countdown	117	0.046	0.0	0.047	0.1
.on_air_monitors.chan-01.countdown	117	0.032	0.0	0.031	0.1
.on_air_monitors.chan-07.countdown	117	0.016	0.0	0.015	0.0
.on_air_monitors.chan-04.countdown	117	0.016	0.0	0.015	0.0
.on_air_monitors.chan-08.countdown	117	0.016	0.0	0.016	0.0
.on_air_monitors.chan-06.countdown	117	0.000	0.0	0.016	0.0
.on_air_monitors.chan-02.countdown	117	0.000	0.0	0.000	0.0
GEnesis_analog_clock_tick	117	1.542	1.3	1.443	2.5
<local>	-	1.113	0.9	1.010	1.7
GEnesis_ticktock	117	0.032	0.0	0.031	0.1
GMS_HighlightEventDisplay	1024	7.240	6.0	6.968	12.1

Figure 6. Hierarchical browser for profile information

- Bindings established to nonexistent text and canvas tags, or to nonexistent binding tags.

The last two are easy to cause by accident, by deleting a canvas item or block of text while neglecting to delete its tags or bindings. These leaks are so difficult to locate because a product such as Purify finds only where they occur in the C source code, and cannot inform the programmer what Tcl code was being interpreted at the time. The Tcl system, moreover, does not export any interfaces that allow the programmer to write code to locate these abandoned items. We have occasionally resorted to the expedient of running the program for a while, stopping it under control of a debugger, and then using the debugger to examine Tk's internal data structures for these items. We do not recommend this activity as an amusing pastime.

6. Conclusions

Using Tcl/Tk has enabled us to develop and integrate systems rapidly. Its unparalleled embeddability has been a major productivity gain, particularly considering that the integration controller, as its name suggests, is intended to integrate the interfaces of video equipment from a number of different vendors.

A couple of worries that were raised during development were easy to address. The first has been a recurring concern that the interpretive nature of Tcl will make the performance of the system unacceptable. Since very little Tcl code is in the direct path of handling device commands and status, this objection really is not an issue. Moreover, the Tcl code in the user interface is a good bit faster than comparable interfaces implemented with toolkits other than Tk, even when the interpreter overheads are taken into account; Tk is *fast*. Another worry dealt with the availability of downstream support. This concern, too, was really not an issue. The source code for Tcl/Tk is freely available, and amounts to only a fraction of the total lines of code that the customer is already committed to maintaining in the GEnesis system. The extensive user community means that support will always be available somewhere.

Several technical issues, nevertheless, give us cause for long-term concern. One of them is stability of the application program interfaces. Each new release has broken our C++ code, and we have had to back and fill to get back on track. In practice, we have skipped every other release or more; we have used, in succession, releases 7.0, 7.4, and 7.6, and will most likely (as of this writing) bypass 8.0 in favor of 8.1. The more widely Tcl/Tk becomes deployed, the more consideration the Tcl/Tk development team will have to give to backward compatibility.

Another concern is the unexplained performance "hot spots". Some operations, such as copying canvases, are simply unusable. The Tcl console, too, becomes unworkably slow if thousands of lines are displayed. The text widget, in general, has to be tuned extremely carefully or it becomes too sluggish for our displays. Some of these behaviors could almost be characterized as "performance bugs" that could be corrected with some development effort, for instance, better indexing to support text tags that include large subsets of the widget.

In spite of these concerns, Tcl/Tk has been a wonderful framework for integrating systems. Each of our worries earlier in the program — for example portability to Windows NT, availability of appropriate database interfaces, and availability of native sockets — was available "just in time" for the next step in development, and we are confident that our remaining concerns will be addressed in the same way.

Acknowledgments

No project of this scale happens without the efforts of dozens of people. In particular, however, the authors would like to thank Steve Zahner, whose early advocacy at NBC was critical to getting Tcl/Tk accepted; David Rabinowitz, who has continued to support our development of the IC to production quality; John Ousterhout, Scott Stanton, Ray Johnson, Brent Welch, and others among the Tcl/Tk development team who freely provided hours of telephone support when we struggled with critical Tcl/Tk bugs; David Henderson, Chris Hammond, Mike Kinstrey, Dana Downey, Joe Amaral, Ralph Minerva, and the other GEnesis developers at GE and NBC.

References

[HH89] Hartson, H. R., and D. Hix. "Toward empirically-derived methodologies and tools for human-computer interface development." *Intl. J. Man-Machine Studies 31* (1989), pp. 477-494.

[HM98] Harrison, M., and M. McLennan. *Effective Tcl/Tk Programming*. Reading, Mass.: Addison-Wesley, 1998.

[KERN95] Kernighan, B.W. "Experience with Tcl/Tk for Scientific and Engineering Visualization." *Proc. Third Annual Tcl/Tk Workshop*, pp. 269–278. Berkeley, Calif.: USENIX, 1995.

[LeDi89] Lehenbauer, K., and M. Diekhans. "TclX - Extended Tcl: Extended command set for Tcl." Unpublished manual, available over the Internet at `http://www.neosoft.com/tclx/man/TclX.n.html`

[LAM95] Lam, I.K. "Designing Mega Widgets in the Tix Library." *Proc. Third Annual Tcl/Tk Workshop*, pp. 53–59. Berkeley, Calif.: USENIX, 1995.

[Poin97] Poindexter, T. "Sybtcl and Oratcl." In "Tcl/Tk Tools", M. Harrison, ed., *Tcl/Tk Tools.* Cambridge, Mass.: O'Reilly, 1997.

[RW96] *Tools.h++: Foundation Class Library for C++ Programming*. Part #RW-30-01-2-032596b. Corvallis, Ore.: Rogue Wave Software, 1996.

[SRY93] Smith, B.C., L.A. Rowe, and S.C. Yen. "Tcl Distributed Programming." *Proc First Tcl/Tk Workshop*, pp. 50-52. Berkeley, Calif.: University of California, 1993.

[SSZ95] Sarachan, B.D., A.J. Schmidt, and S.A. Zahner. "Prototyping NBC's GEnesis Broadcast Automation System in Tcl/Tk." *Proc. Third Annual Tcl/Tk Workshop*, pp. 251–260. Berkeley, Calif.: USENIX, 1995.