Tcl and Concurrent Object-Oriented Flight Software: *Tcl on Mars*

David E. Smyth
david@devvax.jpl.nasa.gov
Mars Pathfinder Flight Software Team
Jet Propulsion Laboratory
MIDCOM Corporation

Abstract

Mars Pathfinder is the first of a new class of "better, faster, cheaper" interplanetary spacecraft missions being developed for NASA at Caltech's Jet Propulsion Laboratory. This spacecraft will be launched during the Mars launch window in December 1996 and will land on Mars in July 1997. The lander carries a small 10kg 6-wheeled robotic Rover that will roam the surface and take samples of soil and rocks.

The flight software for Mars Pathfinder has a concurrent object-oriented architecture, using many concepts adapted from the Actor school of thought, as espoused by Carl Hewitt, Gul Agha, and others. The flight computer, a 22 MIP radiation hardened derivative of the PowerPC has 128 megabytes of RAM and uses the VxWorks real-time POSIXish operating system. Subsystems (each of which is an object, and contains and/or controls other objects) executes in one or more threads. Each object is event and message driven.

This paper describes the current early development effort, where we are using Tcl and its object oriented extension itcl, combined with tclX, blt, and tk, as the language for inter-object messages, for the monitor and control environment, and for the initial implementation of several flight software responsibilities. As the system develops, the flight software may remain as Tcl, or it may evolve into C. The similarity between Tcl and C makes the translation of objects from Tcl to C reasonably straightforward.

Overview

The Mars Pathfinder flight software is being developed as concurrent object-oriented software. Objects with extensive collaborations are grouped into subsystems.

Each subsystem instantiates its own Tcl interpreter, and provides its set of commands that can then be used to invoke the methods of the various objects within the subsystem. Each subsystem has a "well known" socket through which it reads Tcl scripts. Most subsystems also use other sockets, timers, and interrupt handlers to interact with spacecraft devices such as the camera, transmitter, and Rover.

In addition, a central interpreter allows commands to be distributed to all subsystems. For historical purposes, we call this central interpreter the "Sequencer" or "Sequence Engine" and scripts interpreted by this central interpreter "Sequences." This terminology is common in the spacecraft industry, but is left over from the days of yore,

when spacecraft were more like Rube Goldberg machines than the interplanetary robots they are today.

Spacecraft Sequencing using Tcl

In the spirit of "better, cheaper, faster" Tcl has been proposed as the sequencing and command language for the Mars Pathfinder flight software. The specific reasons include:

1. It is already defined. No effort, cost, nor schedule are required to define, document, develop flight software, adapt ground software (SEQ, SEQTRAN and CMD) and test a sequencing and command language.

ð *Faster k Cheaper*

2. Tcl defines only these simple but general concepts:

- A simple syntax: Tcl consists of statements, each statement having the form:
  keyword <parameter ...>
- a couple mechanisms to group parameters (curly braces and quotes)
- the concept that "statements" can return string values
- a way to indicate one wants the result of a statement (brackets)
- global and local variables, and a way to indicate one wants the value of a variable (dollar sign)
- The itcl extension also provides a simple and useful object model.

Tcl itself defines nothing else. Typically, Tcl "registers" procedures to implement what one would normally consider the "reserved" words of the language, including case, for, while, and if. Even set (for assignment) and proc (for defining Tcl procedures) are registered procedures: they are not part of the syntax of the language.

Therefore, we can tailor Tcl to include only those capabilities we really want aboard the spacecraft. All superfluous keywords can be deleted to save memory, or can be added later for additional capability.

ð *Elegance is Better*

3. Since Tcl is all ASCII, we don't have to also define an ASCII to binary translation for sequences: the SEQTRAN ground software system does not need to be adapted for Mars Pathfinder, as it does with other JPL missions. The CMD ground software system can provide the data envelope required by DSN with little or no project specific adaptation.

ð *Cheaper*

4. Tcl is scalable: from 11k bytes for the basic kernel to 200k bytes for a full UNIX shell.

ð *Smaller is Better*

In the most desirable situation, where we never need any logic nor variables in sequences, we only need the absolute "core" capabilities of Tcl. Tcl is designed and implemented such that we can fly a Tcl consisting only of two small files:

```
tclBasic.c
   609 lines of C
   bytes compiled:
   4384 text, 1568 data

tclParse.c
   631 lines of C
   bytes compiled:
   4688 text, 568 data

Total:
   1240 lines of C
   bytes compiled:
   9172 text, 2136 data
```

These files do the following (extracted from the C source header):

```
/*
 * tclBasic.c --
 *
 * Contains the basic facilities
 * for TCL command interpretation,
 * including interpreter creation
 * and deletion, command creation
 * and deletion, and command
 * parsing and execution.
 */

/*
 * tclParse.c --
 *
 * This file contains a collection
 * of procedures that are used to
 * parse Tcl commands or parts of
 * commands (like quoted strings or
 * nested sub-commands).
 */
```

5. Tcl allows us (the Mars Pathfinder Flight Software Team) to act confident in claiming that all we need to provide are the high level commands which don't need any logic, because we know that we have in our back pocket the ability to upload arbitrarily complex scripts!

ð *CYA is Better*

6. If we discover, next year, that we need some more capabilities for our sequencing system, like logic and event triggers, then there is no cost: Tcl already does this. We simply leave these capabilities installed.

ð *CYA is Better*

Tcl as Object Communication Language

Besides ground control of the spacecraft as a system, it is also necessary to control individual software components (threads or Subsystems). Tcl, passed via message queues (specifically, VxWorks pipes), is proposed as the mechanism used to control the flight software components. Therefore:

- Sequences consist of Tcl scripts
- High level commands are names of on-board Tcl scripts.
- On-board software components pass Tcl scripts to each other to implement inter-object messaging

The advantages of the proposed approach include:

1. We will need some form of scripting system in order to do testing, especially regression testing. Tcl has been proven to be effective as a scripting language for regression testing of multi-threaded software, and the Mars Pathfinder Flight Software will be multi-threaded. By using Tcl as the mechanism for all inter-software communication (i.e., between the ground and the spacecraft, and between the software subsystems), then we get a free mechanism for running regression tests. This also means that all development and testing use the “standard” or flight interfaces,

thereby better demonstrating capabilities at an earlier date.

ð *Better, Faster k Cheaper*

2. Using Tcl for our inter-subsystem communication means we don't need to define and implement something new -- this is identical to the sequencing problem.

ð *Faster k Cheaper*

3. Using Tcl for our inter-subsystem communication makes it easy to pass such information across message queues. The ASCII representation insulates components from each other's completeness (a command can be safely passed even if the recipient has not yet implemented it) and many data type concerns (numeric types can be changed to any other numeric type without interface concerns). This allows different parts of the flight software to evolve at different rates -- and it certainly will!

ð *Faster Development is Better*

4. Using Tcl for our inter-subsystem communication makes it easy to perform unit testing: a unit test includes the Tcl scripts (commands) the unit expects to get from other subsystems, and generates Tcl scripts (commands) which it should be sending to other subsystems. Tcl is standard text, so the output is easy to validate.

ð *Better Testing*

5. Using Tcl as the external control for any given subsystem makes it trivial to pass "low level commands" via the same ground environment. We will want to do this during development, and we might want to have this capability for contingencies. Using Tcl adds significant flexibility at zero cost during development and operations.

ð *Better Flexibility*

6. Using Tcl as the external control for any given subsystem enables us to provide behaviors as scripts. I agree that this is very optional: it is always possible to provide different behaviors all in C, and select between which one is active via a switch. However, implementing independent, primitive behaviors in C and then implementing higher level, complex behaviors in Tcl can make it easy to upload "software" with less "political" ramifications: we don't need to uplink binaries, only textual scripts (i.e., Sequences).

ð *Better Flexibility*

7. Tcl and C are very similar languages in look and feel: therefore, we don't need to sweat the limits: what is done in C, and what is done in Tcl (i.e., via Sequences). We can easily move this "limit" over time without big impacts. We can start out writing many things in Tcl, and evolve the system until very little is in Tcl.

ð *CYA is Better*

The NIH Factor: Why Tcl?

There are many different possibilities for languages. In the past, JPL has developed languages for commanding and sequencing

gspacecraft as a matter of course. Many of the advantages of using Tcl can also be achieved by developing a special purpose language here at JPL. The question of why, then, we should use Tcl has arisen. The reasons for Tcl include:

1. Developing languages is an iterative process: they often start out elegantly and devolve into a nightmare (e.g., C++), or they start out being ugly and slowly evolve into something reasonably nice (Fortran90). This process is both slow and expensive. Tcl has been tested by fire literally around the world. People like it, even if they have no vested interest in the language. That is quite rare (consider Ada and C++ as examples). I doubt that we will trivially invent something better. Yet using Tcl *is* trivial.

2. We can say that we are done with the sequencing language already. We can honestly say "yes" to any question anyone asks about its capability ("Can you compile it?" Or "Can you write loops?" Or "Can you do things based on time?" Or "Can you utilize the VxWorks capabilities?" Or ...).

MVC and Tcl

MVC, or Model-View-Controller, is a useful paradigm for software. For spacecraft flight software, it is especially useful.

First, let's establish the meaning of MVC. Any given software object can be considered to have three aspects: The Model aspect is the inate characteristics and behavior; The View aspect is that which is visible about the object, how it looks, how it shows its inate characteristics to the outside world (people or other objects); The Control aspect is that which allows the object to be manipulated, technically, how the object's methods are invoked.

A graphical user interface is ideal for viewing and controlling an object. Often, an MVC programmer implements mechanisms to view an control an object using various mechanisms provided by a GUI environment, and then implementes the "model" or innate object code using some general purpose programming language which can be easily connected to the GUI environment.

On Mars Pathfinder, we start out by implementing objects using the itcl object-oriented extension to Tcl. The view and controller aspects are then directly coded, also in Tcl, using the X Window Widgets provided by Tk. A single object, for example, the Packet Buffer, contains the model, view, and control aspects all jumbled together.

This set of itcl objects can be easily executed on our development workstations, and demonstrated to "customers," in our case, managers, the scientists, spacecraft systems engineers, and the spacecraft operations staff.

Working with these itcl objects, we can evolve the behavior and collaborations between the objects rapidly and conveniently.

Once the objects seem to work pretty well, then we can start evolving them into flight software.

The first step in the evolution is to revisit the objects, and rip each one in two, resulting in one object which provides the "Model" or innate capabilitiy of the object, and another which provides both the View and Control aspects. The "Model" object is the prototype for the actual flight software, while the "View-Control" object remains workstation software.

The View-Control object provides a GUI for the flight software. It is used to control and monitor the software during testing. We expect that the same software will continue to be useful for monitoring the software during flight. However, the 20 minute light-time delay between Mars and Earth will probably prevent us from using these objects to control the flight software during the actual mission.

One of the primary reasons Tcl was chosen for this task was that local and remote transfers of control require only trivial transformations of Tcl code. Therefore, ripping the objects apart so some method invocations may be across the net is usually very simple.

Nevertheless, some changes to the methods do occur, because state data must now be transferred between the two parts: in the all-in-one object, the View widgets can often directly display information as it is updated by the Model methods. Once split, the Model methods must explicitly transfer the changed information back to the View-Control part in order to display data.

Initially, we can run the Model object on a flight-like-test-system running the VxWorks operating system somewhere over the net. The View-Control software running on the development workstations can use "expect" to rlogin to a VxWorks host and start a Tcl interpreter and load the Model objects as flight software.

Another primary reason for using Tcl is that Tcl and C are reasonably similar in structure. The object-model supported by itcl is easily implemented in C, using the guidelines from the author's paper on Object-Oriented Programming in C. Therefore, we can start by running the Model objects as itcl objects under a Tcl shell running on the VxWorks target. We can then evolve the itcl methods into C code, if needed, as needed.

Summary

Using Tcl and MVC concepts during prototyping and developing allows us to provide rapid prototypes for requirement refinement, while giving us a simple path for evolving the actual flight software directly from the prototype. We get the user interfaces we need to monitor the spacecraft, and a testing environemt, essentially for free.

Using Tcl gives us more capability than we hope we need for sequencing, but it also gives us necessary capability for testing, and desirable capability for inter-

subsystem communication. And, it covers our ass for free.

Essential Reading

The Mars Pathfinder Mosaic Home Page URL http://thor2.jpl.nasa.gov.

Papers you find using http://cs.indiana.edu/cstr/search using keywords such as "actor concurrent object"

Gul Agha, "Actors", MIT Press, 1986.

Agha, Wegner, and Yonezawa (eds) "Research Directions in Concurrent Object-Oriented Programming" MIT Press 1993.

Brad Cox, "Object-Oriented Programming: An Evolutionary Approach." (the first 30 pages). Addison-Wesley.

Don Libes, "expect: Curing Those Uncontrollable Fits of Interaction." Summer 1990 USENIX Conference.

Michael J. McLennan, "[incr Tcl] - Object-Oriented Programming in Tcl," the file Intor.ps with the itcl distribution.

John Osterhout, "Tcl and the Tk Toolkit" Addison-Wesley 1994.

Wind River Systems "VxWorks Programming Guide"

Author

David E. Smyth received his B.S. in Computer Science from the University of California at Irvine. His software flew aboard EWACS since 1976, aboard the Space Shuttle since 1980, and aboard the Starship Enterprise from 1979 until the last mission of Star Trek: the Next Generation (he wrote the video editor software). He has implemented systems for flight control, management information, robotic control and navigation, and an object-oriented UNIX. He has presented papers and/or tutorials on real-time software, fault tolerant software, object oriented software, and GUI software in all 4 time zones of the US, 3 time zones of Europe and the Middle-East, and in China. He is perhaps most widely known as the author of Wcl, the Widget Creation Library, which is useful for developing Motif, Athena, and OpenLook GUIs for X Window workstations.

Mr. Smyth is married with three children. He has sailed yachts across the Atlantic twice, transpacific three times, with extensive voyages in Californian, Caribbean, Latin American, European, and Hawaiian waters. Regardless of his hair, he can barely play electric guitar.