Combining Fortran and scripting languages

Arjen Markus[1]
WL | Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

Introduction

Combining Fortran and C in one program may seem like a daunting task sometimes, but if you stick to a few simple rules and use either the proper tools (such as "cfortran.h", *cf.* Burow, 2001) or know the details of your compilers, it is relatively easy. Still, you should do it only if you have good reasons for it. Such reasons might be:

- You have a graphical user-interface written in C and need access to an existing Fortran library for, say, numerical tasks.
- You have a Fortran program to do all kinds of wonderful things, but one part involves handling regular expressions. The easiest then is to interface to a C library that handles such expressions.

This paper focuses on the latter type: a Fortran program that uses a well-written C library to achieve the desired effects. These effects may seem a bit peculiar: we want to have a very flexible way of reading input and writing the output. The C library being used is that of the Tcl/Tk scripting language (*cf.* B. Welch, 2000 or J. Ousterhout, 1998). As this language, Tcl stands for Tool Command Language, was first designed for incorporating it in programs, using it in C programs is easy. The "novelty" comes from using it in a Fortran context. (Note that it is also possible to go the other way around: accessing C and Fortran library from Tcl programs or *scripts*. This is in fact a quite common way of doing things.)

Why should one bother with a scripting language? Well, these languages, and Tcl is no exception to this, offer a flexibility that is very hard to approximate in compiled languages. Typically a program or script can be run without (explicit) compilation and errors are detected and reported rightaway, which makes the development cycle short and tight. They also offer such features as matching strings against regular expressions, communicating via sockets, accessing databases and building graphical user-interfaces with code that is almost or even entirely platform-independent. The rest of this paper will refer to Tcl and its specific properties (*cf.* Appendix A which presents an overview).

Input and output via Tcl

One thing that Tcl could be used for in a Fortran program is input. Even though Fortran's capacity to read plain files is adequate or even excellent in many cases, it does require a lot of work to make it robust. Take for example the following input (where the order of the lines does not matter):

```
number-steps   10
stepsize        0.1
```

If we read this in Fortran, we need to scan the line for a keyword and a value. The value must be stored in the variable belonging to the keyword. A not so robust piece of code to do this is:

[1] E-mail address: arjen.markus@wldelft.nl

```
character(len=40) :: keyword, value
integer           :: number_steps
real              :: stepsize

do
   read( 10, *, end = 100, err = 100 ) keyword, value
   if ( keyword .eq. 'number-steps' ) &
      read( value, * ) number_steps
   if ( keyword .eq. 'stepsize' ) &
      read( value, * ) stepsize
enddo

100 continue
    close( 10 )
```

This code is not robust, because it does not check whether there are both a keyword and a value on the line. It also does not check if there is more information (which might have to be considered an error) or check that the value is of the proper type.

In Tcl, however, the input can be viewed as code – we merely need to define two routines, called *number-steps* and *stepsize* that each take one argument and store that in the right variable:

```
proc number-steps {value} {
   global number-steps
   set numbers-steps $value
}
proc stepsize {value} {
   global stepsize
   set stepsize $value
}
```

With a statement like:

```
if { ![string is double $value] } {
   error "Value is not a valid real"
}
```

the routine can check that the value is indeed of the expected type.

Reading the input then becomes a single statement, namely "sourcing" a script:

```
source "values.inp"
```

Mismatches in the number of arguments are all taken care of automatically by the Tcl run-time system:

```
number-steps 10 20

==> called "number-steps" with too many arguments
```

If we define the input slightly differently, then the input file becomes regular Tcl code without us having to define even these procedures (we do lose explicit type checking then):

```
set number-steps   10
set stepsize        0.1
```

This input file uses the Tcl command "set" and is thus valid Tcl code which assigns values to the two variables.

By structuring the input as valid Tcl code, we can also document the input via comments without having to program a single statement:

```
#
# Define the number of steps for the calculation and the
# step size to be used.
#
number-steps   10
stepsize        0.1
```

In a similar way, output in Tcl is very easy: whether writing to a file, to a pipe or to a socket, you use the same commands, *puts* and *write*, for formatted or binary output. Thus, it depends exclusively on the open statement where the output is going to:

```
#
# Choose a suitable output medium
#
if { $use_socket } {
   set channel [socket $host $port]
} else {
   set channel [open "calc.out"]
}
puts $channel "Output commences …"
```

Using Tcl in a Fortran program

A disadvantage of scripting languages is that they can be slow when dealing with large-scale calculations (though they are quite suitable for mathematical applications as defining and solving ordinary differential equations, *cf.* Markus, 2002). This is, of course, one area where Fortran is very suitable. By combining both types of languages we can achieve the best of both worlds: flexible handling of input and output and robust and fast computations.

This section presents a working example of the use of Tcl inside a Fortran program (it is also possible to use a Fortran library inside a Tcl program – this is in fact greatly simplified by the use of the proper tools, *cf.* Landers, 2002, but would require a different approach to the setting up the application.).

To apply Tcl within a program you need to interact with the run-time system, the Tcl interpreter. This is actually a collection of variables and Tcl routines. Most commonly there is only one interpreter, but there are classes of applications where several are used at once, as each may act quite independently, for instance in client/server systems.

I created a small library to facilitate the interaction:

- One routine, *ftcl_start*, to initialise the one interpreter that will remain active during the whole program run. It can optionally run a startup script from file.
- A set of routines to set the values of variables in the interpreter (one for each common data type, but via the interface mechanism in Fortran, they present themselves as a single routine, *ftcl_set*)
- A set of routines to get the values of variables in the interpreter (likewise, externally they are visible as a single routine, *ftcl_get*).
- One routine to evaluate Tcl scripts (one or more commands; *ftcl_script*)

With this library I created a small demonstration program, an almost trivial simulation, that provides on-line visualisation of the results:

```
! Simple program to show the Ftcl library
!
program calc
```

```
   use FTCL
   implicit none

   integer       :: i
   integer       :: nosteps
   real          :: x
   real          :: y
   real          :: dphi
   real          :: phi

!
! Start the Tcl interpreter, and read the major parameters
!
   call ftcl_start( 'calc.tcl' )
   call ftcl_get( 'dphi', dphi )
   call ftcl_get( 'nosteps', nosteps )

!
! Run the calculation and output the variables
!
   do i = 0,nosteps
      phi = real(i) * dphi
      x = cos( phi )
      y = sin( phi )
      call ftcl_put( 'x', x )
      call ftcl_put( 'y', y )
      call ftcl_script( 'transfer' )
   enddo

   call ftcl_script( 'close_transfer' )
   stop
   end
```

The interesting feature here is that the Fortran program does not need to know anything about the output mechanism – this is all put into the Tcl routine *transfer*.

The script file that is run when initialising the interpreter looks like this:

```
# Define the routines to send data to the server
#
proc transfer { } {
   global channel
   global x
   global y
   puts $channel "$x $y"
   flush $channel
}
proc close_transfer { } {
   global channel
   close $channel
}

#
# SetUp accepts zero or one arguments, if there is no argument,
# use the local host.
#
proc SetUp  { {host localhost} } {
   global channel
   set port    8085

   set channel [ socket $host $port ]
}

SetUp

#
# Set the computational parameters
#

set nosteps 100
```

```
set dphi    0.1
```

The routine *SetUp* sets up a socket connection to the local host (as there is no host name given). The routine *transfer* writes the values of the variables x and y to the channel and flushes it to make sure the output is available immediately on the receiving side.

The receiving side is a somewhat less simple Tcl script that uses the graphical toolkit, Tk, to display the input (the x and y coordinates) graphically:

```
#
# SetUp the server side (it needs to "listen" to the port)
#
proc SetUp  { } {
   set port    8085
   set timeout 60000

   socket -server [list Accept $timeout] $port

   # We run in a graphical (Tk) shell, so an event loop is
   # already available: the next statement is not required.
   # vwait forever
}

#
# Procedure that accepts the client and sets up the connection
#
proc Accept { timelimit socket ip args } {
   fconfigure $socket -block false
   fileevent  $socket readable [list DrawInput $socket]
}

#
# Draw the input graphically
#
proc DrawInput { socket } {
   global xprev yprev

   if { ! [eof $socket] } {
      gets $socket line
      set x [lindex $line 0]
      set y [lindex $line 1]

      if { $x != "" && $y != "" } {
         set xc [expr 100+80*$x]
         set yc [expr 100+80*$y]
         .c create line $xprev $yprev $xc $yc -fill black
         set xprev $xc
         set yprev $yc
      }
   }
}

#
# Main code: create a window in which we can draw and start
# the server …
#
global xprev yprev

set xprev 0.0
set yprev 0.0

canvas .c -background white
pack .c

SetUp
```

The server's version of *SetUp* creates a so-called server socket and then enters an event loop (explicitly via the command *vwait* or automatically because the runtime environment is

graphical). The other routines have to do with the handling of incoming requests and incoming data. The result, a very simple picture, is shown in Figure 1.

(Figure 1)

Conclusion

Combining Fortran and other languages into one program may be cumbersome, but with the right tools it is possible to achieve very rewarding results. With a mere 100 lines of code, we have created a connection between two programs that can run on two completely different machines, where the one is showing the results of the other.

Literature

B. Burow (2001)
cfortran.h, Interfacing C or C++ and FORTRAN
http://www-zeus.desy.de/~burow/cfortran

C. Flynt (no year)
TclTutor
http://www.msen.com/~clif/TclTutor.html

S. Landers and J.C. Wippler (2002)
CriTcl - Beyond Stubs and Compilers
Ninth Tcl/Tk Conference,
Vancouver, september 2002
http://www.digital-smarties.com/Tcl2002/critcl.pdf

A. Markus (2002)
Doing mathematics with Tcl
Third European Tcl/Tk User Meeting,
Munich, june 2002
http://www.t-ide.com/tcl2002e/mathematics.pdf

J. Ousterhout (1998)
Scripting: higher level programming for the 21st century
IEEE Computer, march 1998
http://www.tcl.tk/doc/scripting.html

B. Welch (2000)
Practical Programming in Tcl & Tk
Prentice-Hall PTR, third edition

Appendix: Overview of Tcl

This summary of the scripting language Tcl (abbreviation of Tool Command Language) is intended to give some understanding of it syntax and semantics, but it is *not* intended as a tutorial (for this purpose there are numerous books and on-line publications available, for instance Welch, 1998, Flynt).

One thing to notice about Tcl is that a script consists almost entirely of function calls or cmmands. Except as a sublanguage in several commands it has no operators.[2] Setting a variable to a certain value is done via:

```
set var 1.0
```

or, setting the variable to the result of another function call:

```
set var [expr 2.0*sin(1.0)]
```

The square brackets cause the Tcl interpreter (or run-time system) to execute the command inside the brackets first and substitute the result.

If a variable name is preceeded by a dollar sign ($), then the value of the variable is substituted:

```
set var2 $var
```

This substitution does not occur inside braces or curly brackets. This essentially means that the evaluation can be postponed until required:

```
if { $var > 10 } {
      puts ″Variable out of range: var = $var″
}
```

Variables have *no intrinsic type*. The same variable can be used to store a string, an integer or floating-point value. Depending on the context, the value will be converted to whatever is needed:

```
set var ″1.0″               ;# The variable var has a
                           ;# string value
set var2 [expr {2.0*$var}]  ;# For the calculation the string
                           ;# value is converted to a number
```

If such a conversion is not possible, then a runtime error follows:

```
set var ″a″
set var2 [expr 2.0*$var]

==> syntax error in expression "2.0*a"
```

[2] The language provides mechanisms, however, to modify the command structure, so that an infix notation like “x = 2 * y” does become possible. It is fairly straightforward, but requires some understanding of the run-time system.

A very important data type is the *list*, a sequence of strings that can be accessed and manipulated separately or as a whole:

```
set listvar {1 2 3 aa b}
set var1 [lindex $listvar 1]     ;# Extract the second element,
                                 ;# counting starts at 0
set var2 [lindex $listvar 2 end] ;# Extract a sublist at the third
                                 ;# element upto and including
                                 ;# the last
```

Lists arise in many contexts within a Tcl script, because the braces turn off variable and command substitution:

```
if { $var < 1.0 } {
   set var2 [expr 2.0*$var]
} else {
   puts "Result is larger than 1.0"
}
```

The *if* command takes two or more arguments, the first is a condition that determines the action to take in the usual way (the condition is evaluated and returns true or false), the second is an action to take when the condition is true, and so on. Because of the braces, the values are *not* substituted immediately and the commands are *not* executed. This is deferred until later, when the strings or, more properly, the lists are explicitly evaluated inside the command procedure.

Another method of structuring data involves *associative arrays*. These are actually collections of key-value pairs. As look-up occurs by means of a string rather than a numeric index, they are very suitable for storing status information:

```
if { $status(flow) == "Running" } {
   …
} elseif { $status(flow) == "Waiting" } {
   …
}
```

The *proc* command creates a new command:

```
proc countWords {line} {
   set count [llength $line]
   return $count
}

set count 0
while { [gets $infile line] > -1 } {
   incr count [countWords $line]
}

puts "Total: $count"
```

Actually, the procedure countWords can be made simpler, because Tcl procedures return the value of the last executed command:

```
proc countWords {line} {
   llength $line
}
```