Generating test programs with TestMake

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

Abstract

Three aspects of Tcl make it a very suitable scripting language for generating (test) programs in, say, Java or FORTRAN 90: its abilities to manipulate long strings or text fragments, its support for associative arrays and the ease with which data can be interpreted as Tcl code. This way, parsing input data is almost trivial.

The application described here exploits these features to generate test programs. The user identifies a piece of code, one or several subroutines for instance or the source code for a whole class that need testing in some isolated environment. The input, as far as the user is concerned, mainly consists of:

- The declarations of input or output variables, possibly with a suitable test to see if the code under test does its job.
- A set of test cases which exercise the code.

It is then the task for the application to generate a complete program from these specifications.

The advantages of this approach are that one is concerned only with the formulation of the test cases (with the help of static analysers even that may be automated to a certain extent) and the evaluation of the results. The details of the program that should run all these tests are taken care of by the general Tcl script.

Introduction

This paper describes a Tcl application that builds (test) programs from straightforward specifications. The idea for creating an application that would generate a complete test program based on some fragments of code developed slowly:

- First of all, Tcl comes with an extensive facility to build and execute test suites. This, however, requires one to specify the outcome of a test as a single string, whereas the results of a (FORTRAN) routine might be an array of real values that can be checked against some criterion.
- Second, building test programs is repetitious and tedious. It means, implementing a series of tests where you have to specify all code to check the result and report about it as well as making sure that all tests are run in the correct way.
- Third, some analysis tools are capable of reporting the conditions by which the flow of control would follow a certain path through the code (see figure 1; ref 1.). This is valuable information when examining the code interactively, but it could be used for generating test cases as well.

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

- The fourth and last source of inspiration was a paper on testing the implementation of POSIX routines on a variety of machines (ref. 2.) In this work, fragments of code were used to construct an almost exhaustive set of test programs by which the various error conditions of a POSIX routine could exercised.

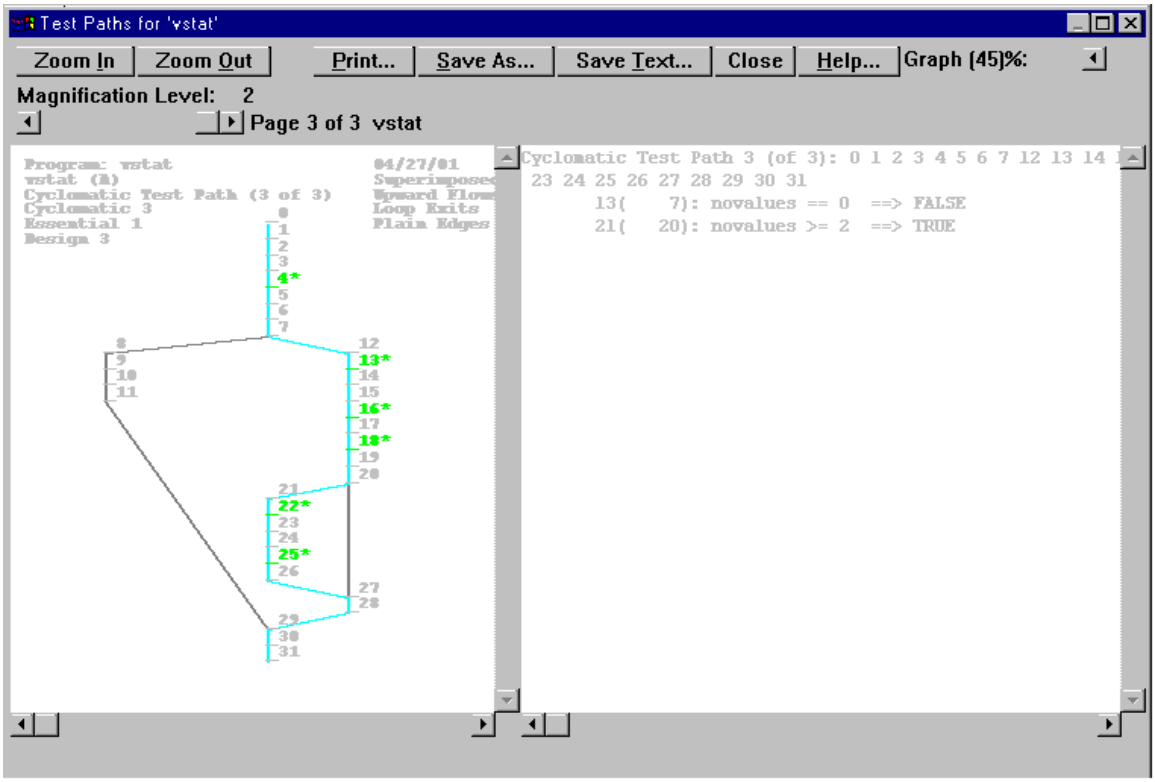

Figure 1. Example of a test path, as generated by a static analysis tool.

A typical situation

To make the idea less abstract, let us examine an example more closely, a hypothetical program that analyses measurement data. Such a program, written in the language of your choice, will have to perform a number of tasks like:

- Gather the input data
- Do the actual analysis
- Report the results

Hence, a coherent part of the program could be a single routine or a set of routines that determine simple statistical parameters (the mean, the extremes, the standard deviation). Such coherent parts can be tested more or less in isolation and that is what `TestMake` tries to facilitate. The routine below in FORTRAN 90 is one implementation[2]:

```
subroutine vstat( values, vmean ,vmin, vmax, vstd )
   implicit none
   real, dimension(:), intent(in) :: values
   real, intent(out)              :: vmean, vmin, vmax, vstd

   integer                        :: i
   integer                        :: novalues
   real, parameter                :: vmiss = -999.0
```

[2] For the sake of brievety all comments have been stripped. There is also no claim to good programming practice.

```
   novalues = size(values)

   vstd = vmiss
   if ( novalues == 0 ) then
      vmean = vmiss
      vmin  = vmiss
      vmax  = vmiss
   else
      vmean = sum(values) / novalues
      vmin  = minval(values)
      vmax  = maxval(values)
      if ( novalues >= 2 ) then
         vstd = sum(values**2) - vmean ** 2 * novalues
         vstd = sqrt( vstd / ( novalues - 1 ) )
      endif
   endif

   return
end subroutine vstat
```

The Java class that follows has the same functionality, although it requires quite a different call sequence:

```
import java.lang.* ;
import java.math.* ;

public class vstat {
   public final float missing_value = -999.0f ;
   private int   novalues ;
   private float vsum     ;
   private float vsum2    ;
   private float vmin     ;
   private float vmax     ;

   public vstat() {
      restart();
   }
   public void add( float value ) {
      novalues ++             ;
      vsum     += value       ;
      vsum2    += value*value ;
      if ( value > vmax ) vmax = value ;
      if ( value < vmin ) vmin = value ;
   }
   public void restart() {
      novalues =  0               ;
      vsum     =  0.0f            ;
      vsum2    =  0.0f            ;
      vmax     = -Float.MAX_VALUE ;
      vmin     =  Float.MAX_VALUE ;
   }
   public float average() {
      return (novalues>0)? vsum / (float)novalues : missing_value ;
   }
   public float stdev() {
      float stdv ;
      if ( novalues > 1 ) {
         stdv = ( vsum2 - vsum * vsum / (float) novalues ) / (float) (novalues-1) ;
         stdv = (float) Math.sqrt( (double) stdv ) ;
      } else {
         stdv = missing_value ;
      }
      return stdv ;
   }
   public float min() {
      return (novalues>0)? vmin : missing_value ;
   }
   public float max() {
      return (novalues>0)? vmax : missing_value ;
   }
} // End of class
```

Inspection of the tasks these *modules* (the term used in `TestMake`, for lack of anything better and less worn out) perform, reveals a number of test criteria:

- The mean and extreme values can be ordered as: minimum ≤ mean ≤ maximum, unless there are no values.
- The standard deviation must be non-negative (otherwise there will be a domain error when the square root is evaluated).
- The statistical parameters can only be determined if there are enough values. Otherwise this must be marked by, say, a reserved value like -999.0.

The modules can be tested with test cases such as:
1. There are no data
2. There is only one value
3. There are two or more different values
4. There are only two measurement data with the same value (which means the standard deviation should be zero!)
5. Other test cases whose expected outcome is easily determined.

With every test case you will want to check the above criteria and perhaps some specific additional conditions as well.

Set-up of the application

The user's perspective

Of course, the application grew more or less organically, rather than from a deliberate design. Nonetheless, we can formulate a number of requirements, based on the likely users. We can not trust formal descriptions to be present and therefore can not rely on a full automation of the task (see below). We can however assume that any programmer who takes the job of testing seriously, can formulate a set of test cases and appropriate test criteria - whether these are sufficient or even suitable, is another matter. A programmer will want to test a reasonably sized part of the whole program, because otherwise formulating the test cases becomes a horrific job. Formulating the test cases and implementing them in a program should be easy to do.

The table below describes these requirements and others in some detail:

Requirement	*Rationale*
Test a reasonably sized part, not just the whole program	Large programs require a large set of test cases. Errors in the program will be difficult to trace because so much code is involved.
Concentrate on the specifications, not on the details of the test program	The test program's details are tedious and lead to errors. Small, clear test specifications reduce the work and the chance for errors.
Conclusions must be clear-cut	The test should either fail (with an indication of what criterion was violated) or succeed. Inconclusive tests are confusing.
Creating the program text should be automated and foolproof	Syntax and other errors in the user-supplied code are acceptable, but the generated code should be error-free.
The specifications should be regarded as data	If the programmer has to *program* the test cases and other information, chances are that he/she makes mistakes. The specifications should be as "data-like" as possible.

Design considerations

A good static analyser is capable of identifying the potential paths of control through the source code and generating a report of which conditions should be met. It is, however, not possible to derive the *test criteria* from this analysis. Consider the following, almost trivial, fragment in C:

```
void Increase( int *value )
{
       (*value) -- ;
       return ;
}
```

Judging from the *name* of the function one would suspect that the decrement ought to be an increment of the variable. But as long as there is no more or less formal description of this function that can be examined by some computer program, we will have to rely on our judgement.

Therefore the input for the `TestMake` application consists of a number of *user-supplied* code fragments that pieced together to form a complete program. The glue for this is provided by a standardised library of fragments, so that the user only has to specify the code that implements a particular test case and subsequently tests the results.

`TestMake` does provide some trivial but important automatic checks. As each parameter that is visible from outside the source code under test, is characterised as *input* or *output* in various flavours, the criteria below can be formulated:

- *Input parameters:*
  Their value must not be changed as a result of invoking the code under test. To test this, a copy is made of their value just before the tested code is run and this copy is compared to the value upon return.
- *Output parameters:*
  Their value must be changed, unless an error condition has been detected by the tested code. Testing that the value has indeed changed can be done in a similar way as for the input parameters, but their initial value must be set to something that is unlikely to be the result of exercising the tested code.
- *Input/output parameters:*
  Some parameters will be updated by the code under test. Thus, they are essentially like output parameters and are treated this way by the automatic testing procedure.
- *Error parameters:*
  It is assumed that an error condition has occurred whenever this type of parameter is set to a different value by the code under test. A correct detection of an error may be part of the test case, so this fact is simply reported and the automatic tests are influenced as indicated above.

In addition to these automatic criteria, the user will want to apply more specific tests. This can be done in two ways:

- As part of the definition of an output parameter
- As part of the definition of a test case

Skeleton code

To generate a full working program, a simple but effective approach is taken in `TestMake`:

- The user supplies the specific pieces of code, such as the declaration of a variable to be watched and the code for the test cases that are to be run.
- A library of small code fragments and auxiliary routines is used as a “glue” for all those pieces.

This way the test program always has the same structure:

```
Main program:
   (Program header)
   Headers
   Declarations of input, output and other parameters
   Additional declarations

   Preparation

   Repeat for all test cases:
      Call initialisation
      Set up the conditions for the test case
      Call the module
      Call the check routine
      Report the conclusions

   End of main program

Subroutine to initialise:
   Initialise all parameters
   Initialisation fragment

Subroutine to call the module:
   Call fragment

Subroutine to check the output:
   Repeat for each parameter:
      Generic check code
      Specific check code (if given)
   Report conclusion: test failed or not
```

By applying a different library (and perhaps redefining a few of the steps) one can generate programs in various languages.

In `TestMake` a whole set of small fragments is distinguished, so that it will not be necessary to redefine the above sequence for each and every language, though the implementation does assume that the programming language (or scripting language) allows for subroutines or methods and for multiple scopes of variables.

Let us examine one item in the construction of the final program more closely, the specification of an output variable. The user will have to supply the name and the type, an initial value (so as to detect the change as a result of the test case), and possibly a check on the resulting value. To take up the example again, the standard deviation might be defined as:

```
Output "vstd" "real" {
       vstd = -1.0 ! Should always be missing value, zero or positive
} {
       call test_failed( .not. (vstd .ge. 0.0 .or. vstd .eq. amiss),
&
              "Standard deviation negative" )
}
```

In the generated program, this information is used to:

- Declare the variable and a *copy* of that variable:

```
real :: vstd
real :: out__vstd
```

- Initialise the variable (and its copy) just before the code under test:

```
vstd = -1.0         ! In subroutine INITIALISE
```

```
out__vstd = vstd    ! In subroutine RUN_MODULE
```

- Check for the changes in the value as a result of the code under test:

```
if ( test_equals( vstd, out__vstd ) ) then
   write( test__lun, * ) "Output parameter vstd has NOT changed!"
   test__output = .false.
endif
```

- Check the value with the user-supplied code fragment:

```
call test_failed( .not. (vstd .ge. 0.0 .or. vstd .eq. amiss), &
       "Standard deviation negative" )
```

In a similar way, all other types of variables to be watched are treated and all code that prepares for the test cases is assembled.

Implementation in Tcl

As stressed in the requirements, the input, as far as the user is concerned, should look like *data* as much as possible. This can be achieved by defining the appropriate Tcl procedures. A bonus of this approach is that the user does not have to know Tcl and the implementation does not have to parse the data - it is simply *sourced*:

- All specifications are contained in an argument to the procedure *Module*:[3]

```
proc Module { module_name definitions } {
   global fragment
   global module
   global module_data

   set module      $module_name
   set module_data data_$module_name
   upvar #0 $module_data a_name
   set a_name(name) $module_name
   set a_name(all)  {}

   set fragment($module,no_testcases) 0

   eval $definitions

   return
}
```

  The Module procedure acts as a container for all specifications. That way it is always clear to what portion (module) of the code a particular fragment belongs.

- Variables are defined using procedures such as *Output*:

```
proc Output { varname vartype initcode { checkcode {} } } {
   global module_data
   upvar #0 $module_data params

   if { [ lsearch $params(all) $varname ] <= -1 } {
      lappend params(all) $varname
   }
   set params($varname,type)      "output"
   set params($varname,vartype)   $vartype
   set params($varname,initcode)  $initcode
   set params($varname,checkcode) $checkcode
   return
}
```

  The element *all* stores the names of all variables defined in this way.

---

[3] The code that is presented here does need some cleaning up.

- For automatically generating test cases from the information obtained by analysis tools, the *Condition* procedure has been defined:

```
proc Condition { expression value exprcode { checkcode {} }} {
   global module_data
   upvar #0 $module_data params

   set params($expression,$value)           $exprcode
   set params($expression,$value,checkcode) $checkcode
   return
}
```

  There should be a specification for both the true and the false value of the condition.

- The user-supplied test cases are specified by means of:

```
proc Testcase { title code { checkcode {} } } {
   global module
   global fragment

   set  no_testcases $fragment($module,no_testcases)
   set  fragment($module,test,$no_testcases) $code
   set  fragment($module,title,$no_testcases) $title
   set  fragment($module,check,$no_testcases) $checkcode
   incr fragment($module,no_testcases)
   return
}
```

  In contrast to the automatic test cases which can be constructed as the file with the information is examined, these "manual" test cases require extra administration, so that they can be recalled later on.

All the basic fragments in the library that provides the glue, are defined using:

```
proc Fragment { codename code } {
   global fragment
   global module

   set fragment($module,$codename) $code
   return
}
```

Because some degrees of freedom are required, for instance to get the names of the variables in, a number of reserved Tcl variables exist. For instance the fragment that declares a variable in FORTRAN 90 reads:

```
Fragment "declaration" {
   $vartype :: $varname
}
```

The variables *vartype* and *varname* are set and expanded when the fragment is written to the source file:

```
proc WriteFragment { codename } {
   global fragment
   global outfile
   global module

   set fragm  \
      "[list subst -nocommands $fragment($module,$codename)]"
   set output [uplevel $fragm]
   puts $outfile $output
   return
}
```

The option *-nocommands* is necessary to avoid conflicting syntax in e.g. Java and C.

The example again

Given the above explanation of how `TestMake` has been set up, let us turn to the Java example again and see how this works out:

- The *preparation* fragment defines a new object that will be used throughout the test suite. The initialisation is meant to reset the status before each test.
- The *initialisation* fragment puts the object in its original state again, something which will be done before each test.
- The *input* fragment for the variable *values* prepares an array of values from which subsets will be used in the actual test cases. This is one way to deal with the need for various test data sets.
- Because the object will accept values one at a time and return the results only via *accessor* functions, the *call* fragment is more involved than its FORTRAN equivalent:

```
Call {
   ! Add the relevant data to the object and get the results
   call vstat( values(first:last), vmean, vmin, vmax, vstd )
}
```

- By setting the first and last indices into the array, each test case defines its own set of test data.
- The example does not include any *condition* fragments, the means to construct test cases from test paths. Currently, `TestMake` is somewhat limited in what it can do there (see the concluding remarks).

The module is specified by the following Tcl code:

```
Module "vstat" {
   Declarations   { vstat v ; int first ; int last ; }
   Preparation    { v = new vstat() ; }
   Initialisation { v.restart() ; }

   Input "values" {float[10]} {
      for ( int i = 0 ; i < 10 ; i ++ ) {
         values[i] = (float) i ;
      }
      values[0] = 1.0f ;
   }
   Output "vmean" "float" {
      vmean = -1.0f ;
   }
   Output "vmin" "float" {
      vmin = -1.0f ;
   }
   Output "vmax" "float" {
      vmax = -1.0f ;
   }
   Output "vstd" "float" {
      vstd = -1.0f ;
   } {
      /* If the standard deviation is not positive, failure! */
      Test.failed( vstd != missing_value && vstd < 0.0f,
                   "Standard deviation is negative"      ) ;
   }

   Call {
      /* Add the relevant data to the object */
      for ( int i = first ; i <= last ; i ++ ) {
         v.add( values[i] ) ;
      }
      vmean = v.average() ;
      vmin  = v.min()     ;
      vmax  = v.max()     ;
      vstd  = v.stdev()   ;
   }
```

```
   Testcase "No data" {
      first = 1; last = 0;
   }
   Testcase "One single value" {
      first = 0; last = 0;
   }
   Testcase "Two identical values" {
      first = 0; last = 1;
   } {
      /* Standard deviation should be zero, otherwise failure! */
      Test.failed( vstd != 0.0f, "Standard deviation should NOT have been zero" ) ;
   }
   Testcase "Two different values" {
      first = 1; last = 2;
   } {
      /* Standard deviation should NOT be zero, otherwise failure! */
      Test.failed( vstd == 0.0f, "Standard deviation should have been zero" ) ;
   }
   # Etc.
}
```

General framework

The Tcl application that is presented in this paper can be regarded as an elaborate example of a whole class of applications. The characteristics of this class are:

- The need or desire to get rid of repetitious and therefore boring and error-prone coding tasks.
- The generation of programs or scripts in some language where the structure of the program is fixed.
- The program to be generated is composed of fixed fragments with only a few degrees of freedom.

Classic examples include, of course, Yacc and Lex, as generators of C programs that fulfil a certain limited task (still others are illustrated by Kernighan and Pike, ref. 3), but one can also think of:

- Simple forms of generic programming, when the programming language itself does not support that directly or only in a rather awkward way.
- XML data handlers - the if-constructs for branching to the proper tag would then be hidden.

Right now, `TestMake` has not been set up with such a more general approach in mind, but its elements could be used to create such a framework.

Concluding remarks

`TestMake` takes advantage of several characteristics of Tcl that make both its implementation and its use easier. With Tcl one can efficiently and effectively manipulate arbitrarily long strings. The input data can be shaped as Tcl procedures, so parsing the input becomes almost trivial. All code fragments are stored and handled via associative arrays. Whereas Tcl is certainly not unique in these respects, it proves a very useful tool.

In its present state, `TestMake` allows for a number of improvements. The script should perhaps generate a makefile for the test program, to facilitate this aspect as well, but right now, it is too restrictive with the construction of test cases from test paths identified by static analysis tools.

Work is now being done to create the more general framework that was mentioned in the previous section.

Literature

1. McCabe & Associates
   User guide to the McCabe Visual ToolSets

2. P. Koopman and J. DeVale
   The Exception Handling Effectiveness of POSIX Operating Systems
   IEEE Software Engineering, volume 26, number 9, september 2000, pp. 837872

3. B. Kernighan and R. Pike
   The Practice of Programming
   Addison-Wesley, 1999