Regression testing of simulation programs

Arjen Markus, David Levelt, Adri Mourits
WL|Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

Abstract
At WL|Delft Hydraulics we develop and maintain large computational programs to simulate (model) the hydrodynamics of such diverse natural systems as rivers, estuaries, lakes and coastal seas. Other aspects of these water bodies that are being modelled in our programs are: water quality in general, primary production (growth of algae), toxicants and morphology (bottom changes due to sediment transport). Because these programs are used by ourselves as well as by our clients, we need to be able to guarantee that new developments and bug fixes do not have unexpected consequences. In other words: we need to verify that results remain the same, unless we know they should be different.

These programs are extensive and complex: partly because they have a long history of development and partly because they are used for many different situations and hence model many physical and biochemical processes. Manual testing new versions became an ever growing burden and several years ago we decided to automate the procedures.

After some research into the various possibilities Tcl was chosen as the language to implement the test suite in. The main reasons for this choice were the ease by which external programs can be controlled and the short learning curve and platform-independence. While over the years the actual software has become quite complex, the original structure and philosophy are still recognizable.

Introduction

Regression testing of large simulation programs is a laborious and time-consuming process: whenever a new version of the program is delivered because of bug fixes or the implementation of new features, you will want to make sure that there are no inadvertent changes in the results. In principle it is a simple black-box testing process: the program or programs that do the simulation are run, the results are compared to the results from a previous, approved, version and depending on that comparison the new version is accepted or not. In practice things are a lot more complicated and this paper describes one solution that we have been using at WL|Delft Hydraulics for a number of years now.

The Delft3D software system consists of a number of individual simulation programs:[1]

- Hydrodynamic simulations for coastal seas, estuaries, lakes and rivers that deal with such diverse phenomena as tidal flow, vertical stratification due to thermal and saline density differences, wind-driven flows and so on.
- Water-quality simulations for studying the spreading of dissolved and suspended substances, algal growth, bacterial pollution and organic micro-pollutants among others.
- Surface wave forecasting and morphological changes due to sediment being transported by the flow and waves, settling and resuspension.

---

[1] http://www.wldelft.nl

All these programs are constantly being developed - to correct errors that were found, to introduce new numerical techniques or the description of new physical and biochemical phenomena. For this extensive system we decided we needed an automated test system.

Design criteria

Because manually testing these programs is very time-consuming and boring, we wanted an automated system. Because there are many individual programs to be tested and all have numerous options, it is impossible to write down a complete test set. So, the system should be extensible, not only by system programmers but more importantly by the scientists who are responsible for the *contents* of the programs. This was an important criterion: to gain confidence, anybody familiar with the use of the programs should be able to add new tests and interpret the results.

The design criteria we came up with are:

- The test suite should run on different platforms (currently: Windows and Linux)
- It should be extensible (adding new test cases should be easy)
- It should be possible for non-programmers to contribute
- It should be flexible enough to cover all our programs
- Results from a new run should be compared *numerically* to those from the reference run
- It should be easy to use:
    - Switch between different version of the computational programs
    - Run only a part of all the tests
    - Allow different levels of details for reporting differences, so that the user can get an overview as well as a detailed
    - Report for specific cases.
    - Both running the test cases and reporting the results should be completely automatic: some of the test cases take several hours to finish. It is then desirable that the test suite runs overnight and the results can be inspected the next morning.

One thing that should be stressed here is that the "testbench" as it is now known does not currently involve the *validation* of the programs: the comparison of the computational results against well-documented experimental data or analytical solutions.

One of us, David Levelt, considered both Perl and Tcl as the implementation language: both are platform-independent, widely available and high-level languages. The advice to use Tcl was based on the simple syntax and the short learning curve.

Problems to be solved

Numerical computations, quite apart from the mathematical aspects, pose a number of challenges to a programmer who wants to check the results:

- Often the output is in the form of binary files. These are relatively compact and easy to read by dedicated programs. For an automated comparison it is however much more convenient to work with "readable" reports: you could use a general tool like *diff* that does not need to know the structure of the data files. At the very least you do not want to extend a dedicated program for each and every test or new test criterion.
- While it is evident for a human being that "1.0" and "0.1e+00" are both representations of the number 1, a program like *diff* was not designed to recognize that. Such different representations of the same number occur when different compilers are used, as with results from different platforms. But even when you try to control the formatting of the numbers,

variations creep in: "+0.1e01" and ".1e001" for instance. This is best solved using a program designed for comparing numbers.

- Perhaps the most challenging problem to be tackled is the fact that computations can be carried out in different ways and this produces different results - even though the program as seen from the source code is correct. Here is a simple illustration:

  $X = 1.0e100 - 1.0e100 + 1.0$

  Because the exact result is much smaller than the individual terms, the computed result in an actual computer program can be either 0.0 or 1.0:

  $X = (1.0e100 - 1.0e100) + 1.0 = 1.0$

  Or:

  $X = 1.0e100 - (1.0e100 - 1.0) \approx 1.0e100 - 1.0e100 = 0.0$

  Of course this is a contrived example, in practice the values are closer together, but the principle remains and a programmer has little or no control over the way the computation is carried out - the compiler may decide to do it one way or the other depending on optimization options or the values left in the cache.

  There may even be "legal" reasons for the variation in outcome: a slightly different formulation of the preceding computation, leading to slightly different operands.

  Again such differences can be handled by a program designed with this in mind: instead of checking for equality, allow a small tolerance. The actual tolerance will depend on what is being computed: the temperature of water in a natural environment is typically from 0 to 40 degrees centigrade whereas the concentration of coliform bacteria near an outfall can be anywhere from 10 to 1 million cells/100 ml or more. So, this is essentially a parameter that must be set by the user in response to the problem at hand.

Two programs are used to solve these problems with the comparison:

- A program to extract the data from the result files, which are mostly in a proprietary format, comparable to NetCDF or HDF and write them out in an ordinary text file for further processing. As this program has a simple command language, extracting the information is easily automated via Tcl.
- A Tcl script to compare two text files line by line. This script reads the lines from the two files that need to be compared and then compares only the numbers they contain:

```
foreach substr1 $line1 substr2 $line2 {
   if { [ string is double $substr1 ] } {
      if { [ string is double $substr2 ] } {
         set diff    [ expr ($substr1-$substr2) ]
         set sum     [ expr ($substr1+$substr2) ]
         set equal   [ expr abs($diff) < $tol2*abs($sum) ]
      }
   }
}
```

In other respects we are luckier: our programs do not depend on complicated databases and they are not integrated with graphical user-interfaces (the GUIs we have are separate programs). So running the programs mostly involves:

- Placing input files in the working directory
- Starting the various computational programs
- Comparing the result files with previous files

We do not have to go through a complicated set of actions to put a database system into a certain well-known state.

Organization of the testbench

Based on the requirements the testbench was set up as follows:

- The testcases are divided into groups - each group emphasizing a particular aspect of the program under test, for instance for the hydrodynamic program: two-dimensional cases, cases with salinity and temperature as additional parameters, cases with various turbulence models. This allows for a simple selection - if only a subset is of interest, then go to the corresponding group of testcases and run only those tests this contains.
- The files belonging to a testcase are stored in a fixed directory structure:
    - For each test case there are general input files and occasionally platform-specific input files.
    - For each supported platform there is a separate set of reference results.
    - The input files are copied to a dedicated work directory and the programs are run from there to ensure the original files are preserved.
    - Dedicated Tcl procedures take care of actually running the programs, catching any output to the standard output and standard error channels.
    - Per testcase the standard "formatted" output files are compared with the reference files and it is also possible to compare specific results from the binary output. As the parameters of interest often depend on the test that is run, this information is stored in a test-specific (script) file that the test designers can edit themselves.

As happens so often, the devil is in the details. Having the whole testbench in a central location like one of our network drives means everybody involved in the maintenance of these programs can access it, but running the tests requires that only one user at a time is active. We could copy the files to a local disk, but the shear size of the testbench makes this impractical - the testbench currently comprises several gigabytes of data and it simply takes too long to copy all these data to be practical. So the original design had to be adapted: the tests can now be run from anywhere, not just from the central repository. This required the use of explicit paths to the executable programs and to the various directories used per test case.

Another aspect that turned out to be a bit cumbersome when the set of tests continued to grow was checking the comparison results per testcase, We needed a very condensed report: one line per case that indicates the success or failure. Of course when a test case shows too large differences vis-à-vis the reference results, it needs to be analyzed in detail. So the full result files must be kept as well.

Further developments

The success of the Delft3D testbench has led to the development of a similar system for another software package, called WANDA. This is a package for hydraulic computations of industrial systems consisting of pipes and pumps. While the basic principle remained the same, the scripts were rewritten from scratch:

- Rather than a grouping of test cases, the developers of the WANDA system wanted a hierarchical structure to organize their cases. The test selected was then to be based on this hierarchy: specify the starting point in the test hierarchy and execute all the tests below that point.
- Having learned from the experiences with the Delft3D testbench, we built in the possibility of running the testbench locally from the start, making for clearer code.
- The WANDA system has only two computational programs and is supported on a single platform only, which made the run procedure slightly easier.

Currently, work is in progress to convert the testbench for a third software system, called SOBEK, to the same framework. SOBEK is a one-dimensional simulation package dedicated to the computation of the flow and water-quality in rivers, canals and urban sewage systems. The original testing system uses a test automation tool for graphical user-interfaces to run the tests and compare the results. However, such a tool is not suitable for this type of work, as the purpose was not to check that the GUIs were operating properly, but to check that the computational results are correct. Any run-time error from the computation program causes the test tool to stop, because something unexpected (or unforeseen by the test scripts) happens. With Tcl however such run-time errors are easily caught and dealt with.

Conclusion

Implementing several automated testbenches in Tcl for the purpose of routinely testing the results from our computational programs against previous versions has made the regression testing a lot easier. Not only can we run tens of tests with a minimum amount of work, the maintainers of the simulation programs do not have to be experts in the implementation language to add new testcases or new comparisons. The framework is flexible enough for them to do it without assistance.

This is greatly helped by several aspects of the Tcl language and run-time system:
- The ease of Tcl's syntax, which means that even inexperienced users can add or adjust test cases
- The flexibility by which one can manipulate files and directories
- The ease with which one starts external programs and catches the output
- The simplicity of the run-time system - users do not need to install the complete Tcl distribution but can use Tclkit for instance

In the time the first Tcl-based testbench has been in use, there has been only one serious problem: the [exec] command fails (on Linux) if the executable program starts a number of threads on different nodes in a Linux cluster. This could be solved by invoking the program via a standard Linux shell (a Bourne shell), but it remains a peculiar issue.

One other area where Tcl may be a less elegant solution is the manipulation of large amounts of numerical data. Such manipulations are important for the *validation* of the simulation results: compare the numerical results to field data and determine a measure of “goodness of fit”. There exist extensions, like NAP by Harvey Davies,[2] to deal with them, but they are not (yet) capable of reading our proprietary data files. At the moment a language such as MATLAB is used to deal with these computations and manipulations.

The most important conclusion is that Tcl has turned out to be a very useful tool for our testing requirements.

---

[2] http://nap.sf.net